University of Amsterdam

System & Network Engineering

Using git to circumvent censorship of access to the Tor network

July 24, 2013

Authors:
Björgvin Ragnarsson bjorgvin.ragnarsson@os3.nl
Pieter Westein pieter.westein@os3.nl

Abstract

The Tor network is designed to provide anonymity and privacy to its users but suffers from censorship from network operators who don't want their users to access Tor. Traffic analysis for censorship purposes can be mitigated by obfuscating traffic and Tor developed a technology called pluggable transports to do so.

We present git-pull-tor, a pluggable transport for Tor that obfuscates the network traffic to resemble usage of the Git version control system. We compare the download speed when using git-pull-tor to downloading files without an obfuscation proxy and find the current version to have considerable performance issues. We find the strength of the system to be its accuracy in imitating Git, since it uses the Git program to perform the obfuscation. The weakness with regard to traffic analysis is that is produces traffic flow patterns that are unusual for Git.

Contents

1. Introduction 1
1.1. Research question . . . 2

2. Related work 2
2.1. Obfuscation . . . 2
2.2. Pluggable transports . . . 2
2.3. Protocol identification and fingerprinting . . . 3

3. Background 3
3.1. Tor anonymity network . . . 3
3.2. Obfsproxy . . . 4
3.3. Git version control system . . . 4

4. Design 6
4.1. Threat model . . . 7
4.2. The network transfer implementation . . . 7
4.3. Obfsproxy as a base for git-pull-tor . . . 8

5. Evaluation 9
5.1. Performance evaluation: Download speed . . . 9
5.2. Security evaluation: Detecting git-pull-tor . . . 10

6. Conclusion 11

7. Future Work 12

8. Acknowledgements 12

A. Performance measurement: Data tables i

B. Client transport script ii

C. Pluggable transport implementation for obfsproxy iii
C.1. Main code . . . iii
C.2. Helper code for encoding/decoding TCP streams in a Git repository . . . v

1. Introduction

Tor is a distributed network of encrypted tunnels for users to route their communications through to provide anonymity. The traffic is encrypted in as many layers as the number of hops it takes through the Tor network, a technique called onion routing [1]. This allows users to use the Internet without their network operator knowing the content or the destination of their communications.

Some network operators do not allow connections to the Tor network and the original method to implement this policy was to block all IP addresses of Tor relays, which are publicly known. This lead to the introduction of bridge relays, which are not listed in the Tor relay directory. Bridge relay operators give out information to users about their bridges in hope that the knowledge doesn't reach the censoring network operators. This has lead to more sophisticated network analysis by censors, using Deep Packet Inspection technology (DPI). The Tor implementation uses OpenSSL[1] to set up a secure TLS connection but the handshake phase of the protocol makes Tor distinguishable from other applications using OpenSSL.

To combat identification of Tor usage, a feature called pluggable transports [2] was developed. Pluggable transports act as SOCKS proxies[2] through which traffic between the client and bridge is directed. Each pluggable transport needs a client and a bridge implementation, where the traffic is obfuscated before it is sent over the network and de-obfuscated at the other end. Support for pluggable transports was added to Tor to decouple protocol-level obfuscation from the core protocol and make it easier for developers to experiment with novel circumvention methods.

This paper introduces a pluggable transport to hide Tor network traffic as being traffic from usage of the Git program [3]. Firstly, Git is a popular version control system amongst developers who share code, often using publicly available servers. This might make the economic impact of blocking Git traffic indiscriminately too high. Secondly, Git is used to transfer data of various sizes and might therefore fit some of different usage patterns of Tor users.

The remainder of this paper, is structured as follows. Section 2 covers the related research on the subject of obfuscation, pluggable transports and protocol identification. Section 3 provides details into the technology used, namely Git and Tor. Section 4 discusses design decisions made for the prototype pluggable transport as well as implementation details. The performance and censorship resistance of the prototype is evaluated in section 5. The paper concludes with section 6 and suggestions for future work are found in section 7.

[1]http://en.wikipedia.org/wiki/OpenSSL

[2]SOCKS proxies forward TCP and UDP packets from clients to servers. They were originally developed for access control and firewalling purposes but can also serve as a circumvention tool.

[3]http://git-scm.com/

1.1. Research question

The main goal of this research is to design and implement a proof of concept pluggable transport which acts like Git. The research questions are the following:

> *Is it possible to shape Tor traffic in such a way that it resembles to the Git protocol? What methods could a censor apply to identify Tor bridge relays communicating with end users using such an obfuscated protocol?*

Additionally, we want to know what the overhead of using git-pull-tor is by measuring throughput and repeat the measurement with no obfuscation.

2. Related work

Previous work has been done in the field of obfuscating Internet protocols and Tor specifically. We provide an overview of some of that work in this section.

2.1. Obfuscation

Obfuscation in general, works toward information hiding with many techniques that are important in a number of areas [3]. The areas that this research is focusing on are **Covert channels** and **Anonymity**.

In covert channels there is a message that someone wishes to send secretly. This message is then hidden within a innocuous message. The hiding process involves a secret-key to restrict detection and/or recovery of the embedded data.

With anonymity, the goal is to hide a users identity, by supplying the user with a fake identity, to be used during Internet activities [4]. Tor extends on the basic anonymity, by addressing the issue of IP tracking, giving users a fake IP.

2.2. Pluggable transports

Appelbaum and Mathewson proposed a protocol for handling Tor obfuscation in a general way, called pluggable transports [2]. Since then, pluggable transport support has been adopted by the Tor software and different implementation have surfaced.

This research centers around mimicry of a known protocol to remain hidden from a DPI systems. **SkypeMorph**, introduced by *Moghaddam et al.* [5], is an example for this approach. It encapsulates Tor traffic into a connection that resembles Skype video traffic. Another example is **StegoTorus**, introduced by *Weinberg et al.* [6], which disguises Tor traffic from DPI systems by resembling HTTP, Skype and other protocols. Both implementations were investigated by *Houmansadr, Brubaker & Shmatikov* who found the approach taken to be flawed [7]. The complexity of protocols such as Skype and inter-dependency of sub-protocols makes it very hard to imitate them. *Houmansadr*

et al. argue that specific implementations of protocols must be imitated if protocol mimicry is to be effective.

Another approach to obfuscating the Tor protocol, is to do so without mimicking a specific protocol. **Dust**, introduced by *Wiley et al.* [8], is such a system. Another example is **ScrambleSuit**, introduced by *Philipp Winter et al* [9]. It is a protocol layer above TCP with the purpose to obfuscate the transported application data.

2.3. Protocol identification and fingerprinting

Protocol identification algorithms, in conjunction with fingerprinting techniques, are widely used to block and filter out unwanted traffic and to restrict Internet access. Encryption mechanisms are often used to circumvent these algorithms, but even then, protocols can be identified by looking at certain parts of the encryption. *Gebski et al.* [10] developed a method, using a graph-comparison approach, to build profiles of several protocols and classify an unknown encrypted protocol against these profiles using only the visible behaviour of the protocol; meaning time, size and direction of the packets.

Besides identifying protocols, due to certain characteristics, fingerprinting flows can be effective as well. *Crotti et al.* [11] present a flow classification mechanism based on three properties of an IP packet; size, inter-arrival time and arrival order. A Gaussian filtering operation is used to derive protocol fingerprints and is effective to deal with noise such as jitter and changes in the packet size.

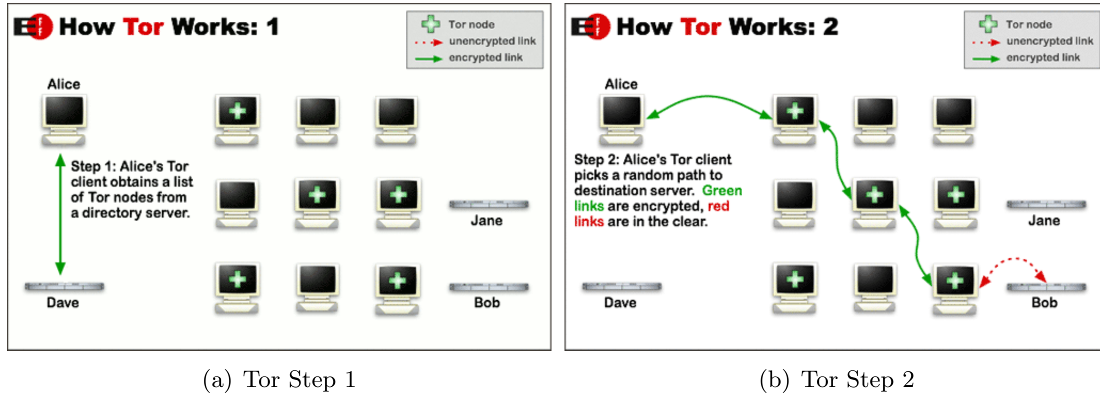
3. Background

Our prototype pluggable transport builds on several technologies. This section provides background information into some of them.

3.1. Tor anonymity network

To access the Tor network, a user needs to perform two steps before sending data. In figure 1(a) the first step is shown. A **directory server** is contacted by the user, to obtain information about the topology of the overlay network. The user will receive a list of nodes, in order to build a circuit. These nodes are known as **relay nodes**. They either act as entry, middle or exit node.

Figure 1(b) shows the second step. Each point in the Tor network represents a node. A minimum of three nodes is needed to build a circuit. The first node, known as the **entry node**, brings the user into the Tor network. Nodes following the entry nodes are known as **middle nodes**, and extend the circuit. The last node in the circuit is known as the **exit node**. This node supplies the user of its IP address and hostname, to be used while accessing the Internet.



(a) Tor Step 1

(b) Tor Step 2

Figure 1: Tor flow diagram: (a) and (b)

To access a website, that is censored in the user's location, the user connects to the Tor network and requests the website. However, the censor connects to the Tor network in order to block Tor nodes. To counter this problem, the Tor project introduced **Tor bridges**. These are unlisted Tor nodes and can be used as entry points to the Tor network. These bridges are not publicly listed, so users need to request a list of bridges manually [12].

3.2. Obfsproxy

Obfsproxy is name of original implementation of the pluggable transports specification. The software has been reimplemented in Python with extendability in mind and functions as a framework for implementing new pluggable transports. Currently there are two pluggable transports implemented, obfs2 and obfs3. Both transports encrypt the traffic using the AES block cipher [4]. Obfs3 improves upon obfs2 by using Uniform Diffie-Hellman (UniformDH) key exchange to establish a shared key [13]. UniformDH uses public keys with properties similar to random numbers, which makes it harder to detect that an obfs3 protocol handshake is taking place.

Like other pluggable transports, obfsproxy functions as a separate application, independent from Tor. Figure 2 shows how Tor connects to obfsproxy, using the SOCKS protocol. Applications other than Tor can also use the obfsproxy framework by communicating with it using the SOCKS protocol.

3.3. Git version control system

Git is a version control system (VCS), originally created for development of the Linux kernel, and has gained widespread use among software developers since its initial release in 2005. Git is distributed, unlike classic centralized VCSes, which means that developers

[4] https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

Figure 2: Obfsproxy diagram. (Source: https://www.torproject.org/projects/obfsproxy.html.en)

have the complete history of tracked content and are able exchange data directly between themselves without being bound to using a central VCS server. The distributed nature of Git is not important for this research and the following assumes a setup with a client and a server.

3.3.1. The object store

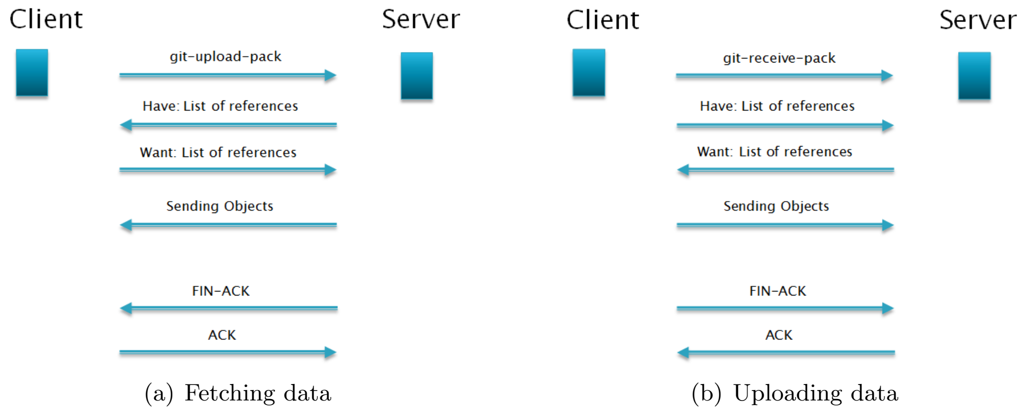
A Git repository functions as a key-value data store. The key is calculated as a SHA1 hash of the data with an additional header. The header specifies the type of object, size in bytes and ends with a null byte. The SHA1 key is used as a file name to refer to the data as stored on disk. Before storing the data, it is compressed using zlib. Figure 3 shows a high level overview of this process.

Figure 3: Storing a file in a Git repository

Git was developed to manage source code and therefore it needed to be able to represent files and directories in its object store. Objects with file content are called **blob** objects. A **tree** object contains a list of SHA1 hash object references, each with an associated file name, type of object (blob or tree) and Unix permissions information. This structure allows file system directories to be represented as Git objects. Other types are **commit** objects which are used to keep track of revisions of trees and **tag** objects which are most commonly used to mark commits as releases of the tracked source code.

3.3.2. The Git transfer protocol

The Git network transfer protocol has two modes of operation, upload and download of objects. Figure 4 shows the interaction between the client and the server for those two operations. Figure 4 (a) describes the process when a client fetches data from the server and figure 4 (b) describes the process when a client uploads data to the server.



(a) Fetching data

(b) Uploading data

Figure 4: Interaction between a client and a Git server

Uploading data

For uploading data to a remote Git repository, the *send-pack* and *receive-pack* processes are used. The former runs on the client-side while the latter runs on the server-side. When the user executes the *git push* command, the *send-pack* process is activated, initiating a TCP connection to the server. This process sends a request to run the *receive-pack* process on the server. The *receive-pack* process responds with a list all references [5] it has. From that information, the *send-pack* process is able to determine what commits are available locally but not remotely and sends them to the remote *receive-pack* process as a compressed file, containing all the objects. Upon receiving, the server responds with a success or failure message.

Downloading data

The download operation is similar as uploading. When the *git pull* command is executed on the client machine, the *fetch-pack* process is activated, initiating a TCP connection to the server. A request is sent to the server, to run the *upload-pack* process on the server which returns a list of all the references available in the repository. The *fetch-pack* process looks at which objects are locally present and responds to the server with the objects that are missing. The server responds by running the *upload-pack* process, which begins sending a file containing all the requested objects.

4. Design

This section starts by defining a threat model and from there, technical decisions for the prototype pluggable transport design are explained.

[5] A reference is a pointer to a commit which is important to keep track of, such as the latest commit.

4.1. Threat model

The adversary (or censor) in the proposed model is an Internet Service Provider (ISP) or a state-level authority with control of the national network. The censor keeps a list of undesired Internet services outside of the network under its control and blocks access to them based on IP addresses of servers and by forging DNS replies. Such undesired services include the Tor relays that are listed in the public Tor Directory. The threat model assumes that the censor does not go as far as white listing acceptable services and block everything else. The maintenance cost of keeping such a list up-to-date and the cost related to delayed access to services in the process of being white listed is assumed to be too high. The censor does not have control over users' computers or software installed on them.

For blocking unknown yet undesired services such as Virtual Private Networks (VPNs) or Tor bridge outside of its network, the censor monitors connections crossing the boundary of its network. The following methods are used to detect connections to undesired services:

Deep packet inspection (DPI) allows the censor to do pattern analysis on application protocols and drop the connection when there is a positive match. This requires predefined rules for each protocol.

Active probing is used when it is not possible to construct a DPI pattern rule that unambiguously detects a connection to an offending service. This can be the case when the DPI software detects a TLS connection but cannot know if the underlying protocol is acceptable or not, such as a connection to a Tor bridge. The censor then connects to the possible Tor bridge and checks if it speaks the Tor protocol.

Statistical analysis such as size of packets and inter-arrival times.

The aim of this project is to obfuscate the Tor protocol to bypass pattern analysis. The strategy applied is mimicking an existing protocol, and the assumption is that the censor does not want to block Git entirely.

4.2. The network transfer implementation

To perform the actual network transfer, we decided to use the Git program[6] to do the heavy lifting and not implement network protocols ourselves. This is done to maximize the correctness of our mimicry and make network traffic analyses harder, as recommended by *Houmansadr et al.* [7]. An additional benefit is that the Git program supports four transport protocol, HTTP, HTTPS, SSH in addition to its own Git protocol.

We assume that the client cannot accept incoming TCP connections, because of a

[6] www.git-scm.com

NAT configuration or otherwise. This means that the client has to initiate sending and receiving of data. The prototype, running on a client machine, comes with a script that runs in the background and periodically sends and receives data from the bridge. The transfer script can be found in Appendix B.

4.3. Obfsproxy as a base for git-pull-tor

We decided to base our pluggable transport on obfsproxy, which is described in section 3.2. We modified the obfsproxy *dummy* transport to our needs. The *dummy* transport does nothing more than passing the TCP stream unmodified between a Tor client and a bridge.

Our use of obfsproxy was a bit unorthodox. Usually, an obfsproxy client makes a TCP connection to an obfsproxy bridge and over that TCP connection they communicate using one of the supported obfuscation protocols. Since we decided to let the Git program take care of the network transfer, we did not need to have obfsproxy set up a TCP connection to the bridge. To simplify prototype development, we decided to use the TCP connection as a 'hello' message from the client, to tell the bridge to start monitoring its Git repository for incoming data. This TCP connection is kept alive without sending any data packets while the obfsproxy client and bridge communicate over Git. This produces a recognizable pattern of a TCP three-way handshake followed by Git traffic on another port and should be replaced in future revisions of git-pull-tor. That could be done by encoding the connection handshake and tear-down in a Git repository, just as currently is done with the TCP stream.

Obfsproxy implements its pluggable transports with the help of Twisted[7], an event-based network programming framework, which has semantics that separate protocols from the underlaying transport layer. The pluggable transport receives the TCP stream from Tor in variable length chunks, and provides the chunk as a parameter to its `receivedUpstream` method which is executed every time such a chunk is received. In the case of git-pull-tor, the `receivedUpstream` method creates a commit in a Git repository that includes only one file with the content of the chunk from the TCP stream. Each commit is linked to the previous one and can queue up in the Git repository for the other end to process. At the same, a thread runs in the background that monitors the Git repository for incoming commits, processes them in a chronological order and executes the `receivedDownstream` method which relays them to the TCP connection to Tor.

The full Python code for our pluggable transport can be found in Appendix C.

[7]https://twistedmatrix.com/

5. Evaluation

Once the prototype was in a working condition, we carried out performance measurements and further security analysis.

5.1. Performance evaluation: Download speed

An experiment was conducted to get insight into how well it performed. An Ubuntu 12.04 server was configured to run Tor version 0.2.4.12-alpha as a bridge. Obfsproxy with git-pull-tor support was installed along with an OpenSSH server. The client was a Lenovo T410 laptop with Ubuntu 13.04, running the same Tor and obfsproxy versions in addition to an OpenSSH client and the curl downloading tool. The transport script from Appendix B was configured to connect to the remote Git repository using SSH.

The experiment measures the time it takes to download a 10 MB file using curl with four different settings:

- Tor was configured to use the bridge and the obfsproxy git-pull-tor pluggable transport. Curl used the Tor SOCKS proxy.
- Tor was configured to use the bridge and the obfsproxy *dummy* pluggable transport. Curl used the Tor SOCKS proxy.
- Obfsproxy was configured to use git-pull-tor pluggable transport. Curl used the obfsproxy SOCKS proxy.
- Obfsproxy was configured to use the *dummy* pluggable transport. Curl used the obfsproxy SOCKS proxy.

Measurements for each of the four settings were repeated 10 times. The results are depicted in figure 5 and the full results of the experiment can be found in Appendix A. It is worth noting that the download failed half of the time for the first setting. The reason for this is unknown but a bug in git-pull-tor could be an explanation.

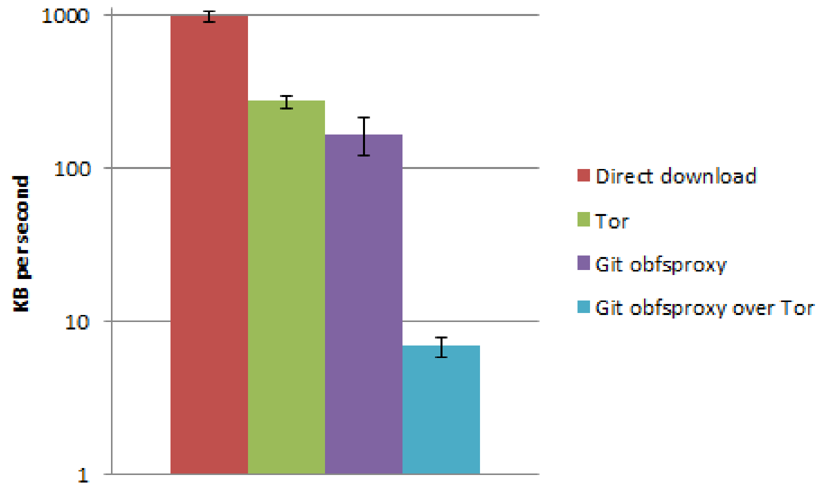


Figure 5: Download speed comparison of different methods of downloading a 10MB file. The current prototype running over Tor is significantly slower than a direct download of a file.

5.2. Security evaluation: Detecting git-pull-tor

Thorough analysis of possible ways to censor git-pull-tor could not be conducted within the time frame of this research. We did, however, compare git-pull-tor to "Requirements for parrot circumvention" as put forward by *Houmansadr et al.* [7]:

Mimicking the protocol in its entirety including side protocols and their interdependence. Houmansadr et al. go further and recommend the approach we took; use an existing implementation which embeds the payload because of complexities involved in recreating protocols faithfully. We selected a simple protocol with only one optional side protocol, DNS. We have not researched if a lack of a DNS request when Git traffic is detected should be considered an alarm in a censoring DPI system.

Mimicking implementation specific artifacts and error handling is done correctly because of the same reason, we use a previous protocol implementation.

Mimicking typical traffic including traffic patterns generated by typical user behavior. We have not researched what can be considered typical Git traffic, neither generally nor withing a specific censored network. We have not researched which versions of the Git programs are popular and should therefore be used. Still, we think is the weakest point of git-pull-tor because the high frequency of git push and pull traffic. The current version of the transport script sends data to the bridge and fetches data from it in repeated cycles which produces very unique traffic pattern.

The following weaknesses of git-pull-tor are also of notice:

The HTTP and Git protocol transport mechanisms don't encrypt the payload. The payload is compressed and a capable censor could decompress it on the fly and do DPI analysis on the content. A censor will probably first try other methods to block git-pull-tor since since decompression adds load to the DPI system. An additional layer of obfuscation which makes the payload look more like content of normal git objects, such as source code.

Payload is written to disk. The current version of our pluggable transport writes the payload TCP stream to Git repositories at both the client and the bridge. It is relatively easy add support for deleting this data but even then it might reveal a lot of forensic information for an investigator who might be able to reconstruct the entire communication between a client and a bridge. This data will encrypted when used with Tor but if git-pull-tor is used as SOCKS proxy to transfer plaintext information such as HTTP traffic, the user's privacy might get completely breached.

6. Conclusion

This paper presents git-pull-tor, a pluggable transport for Tor that disguises communications as Git traffic. As with other circumvention technology, git-pull-tor will work as long as it is not being targeted censors. Once the decision has been made to try to block it, time and resources are required.

A key feature the use of an existing program to transfer the payload traffic, and thus it limits the possibilities of fingerprinting based on inaccurate implementation of the network protocol. However, the usage of the protocol can look suspicious in the eye of a censor as continual polling is unusual for Git usage such as software development.

Performance measurements show that the prototype offers lower throughput than the use of plain Tor. However, the bit rate is enough to facilitate browsing text-based or low-visual-enhanced websites. This could be improved with further optimizations.

7. Future Work

The current prototype of git-pull-tor is in its infancy. Future development and discussion of its viability will take place at: https://trac.torproject.org/projects/tor/ticket/9192

The following are ideas for additional features to the pluggable transport, in case it will be included into the obfsproxy framework.

Eliminate disk writes. As mentioned in section 5.2, disk writes are unwanted in certain situations. If the Git program would be modified to work with repositories in memory, these disk writes would be eliminated.

Make use of public Git servers for relaying traffic. Users can be blocked from directly accessing bridges but still be able to connect to public Git hosting providers. This could be used for relaying the traffic between the user and the bridge. This comes with additional performance issues and the same disk writing problems as previously discussed.

Layered obfuscation. An additional layer of obfuscation could be useful if censors start uncompressing the HTTP & Git protocol traffic for identification. This layer could be in the form of encoding the payload to look like source code or other acceptable data.

8. Acknowledgements

We would like to thank our supervisor, Henri Hambartsumyan, for his feedback, support and review of the paper. We would also like to thank the rest of the team at Deloitte Risk Services Netherlands for their input, in particular Daan Muller for reviewing a draft version.

References

[1] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing for Anonymous and Private Internet Connections. *Communications of the ACM*, 42(2):39–41, 1999.

[2] Appelbaum J. and Mathewson Nick. Proposal: Pluggable transports for circumvention. `https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/180-pluggable-transport.txt`. Accessed: 2013-06.

[3] Fabien AP Petitcolas, Ross J Anderson, and Markus G Kuhn. Information hiding-a survey. *Proceedings of the IEEE*, 87(7):1062–1078, 1999.

[4] Jacob Palme and Mikael Berglund. Anonymity on the Internet. *Retrieved August*, 15, 2009.

[5] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. Skypemorph: Protocol Obfuscation for Tor Bridges. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012.

[6] Zachary Weinberg, Jeffrey Wang, Vinod Yegneswaran, Linda Briesemeister, Steven Cheung, Frank Wang, and Dan Boneh. StegoTorus: A Camouflage Proxy for the Tor Anonymity System. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 109–120. ACM, 2012.

[7] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. The Parrot is Dead: Observing unobservable network communications. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

[8] Brandon Wiley. Dust: A Blocking-Resistant Internet Transport Protocol. 2011.

[9] Philipp Winter, Tobias Pulls, and Juergen Fuss. ScrambleSuit: A Polymorph Network Protocol to Circumvent Censorship. *arXiv preprint arXiv:1305.3199*, 2013.

[10] Matthew Gebski, Alex Penev, and Raymond K Wong. Protocol identification of encrypted network traffic. In *Web Intelligence, 2006. WI 2006. IEEE/WIC/ACM International Conference on*, pages 957–960. IEEE, 2006.

[11] Manuel Crotti, Maurizio Dusi, Francesco Gringoli, and Luca Salgarelli. Traffic classification through simple statistical fingerprinting. In *ACM SIGCOMM Computer Communication*, volume 37.1, pages 5–16. ACM, 2007.

[12] Gibson Aaron. The Tor Project: The bridge distribution database. `https://bridges.torproject.org/`. Accessed: 2013-06.

[13] Dan Boneh. The decision Diffie-Hellman problem. pages 48–63, 1998.

A. Performance measurement: Data tables

	obfsproxy with dummy	**Tor**	**git-pull-tor**	**git-pull-tor over Tor**
Round 1	9,148	41,397	48,439	1149,024
Round 2	8,897	33,579	46,051	failed
Round 3	10,36	36,296	149,709	1634,239
Round 4	9,415	34,494	55,331	1397,613
Round 5	12,392	37,71	69,373	1432,368
Round 6	9,944	39,396	45,571	failed
Round 7	10,253	39,268	50,99	failed
Round 8	10,376	30,968	69,997	1656,274
Round 9	10,248	38,3	72,127	failed
Round 10	9,746	31,912	52,733	failed
Average	10,0779	36,332	66,0321	1453,9036
Standard Deviation	0,966658557	3,475338033	31,11076262	206,1908215

Table 1: Duration of downloads in seconds: Download using a dummy obfsproxy, download over Tor, git-pull-tor and git-pull-tor over Tor.

B. Client transport script

```
#!/bin/bash

REMOTE=user@bridge.example.com:/tmp/gittor
GITDIR=/tmp/gittor

git --git-dir=$GITDIR remote rm origin
git --git-dir=$GITDIR remote add origin $REMOTE

git --git-dir=$GITDIR prune

while true;
do
  git --git-dir=$GITDIR \
    push  -f origin refs/heads/*:refs/remotes/origin/*
  git --git-dir=$GITDIR \
    fetch -f origin refs/heads/*:refs/remotes/origin/*
done
```

C. Pluggable transport implementation for obfsproxy

C.1. Main code

```
""" This module contains an implementation of the 'git' transport. """

from twisted.internet import reactor
import obfsproxy.common.log as logging
from obfsproxy.transports.base import BaseTransport
from obfsproxy.network.gitbuffer import GitBuffer
import string, random, thread, threading

log = logging.get_obfslogger()

gitPollingThreadLock = threading.Lock()
KeepGitPollingAlive = False

class GitTransport(BaseTransport):
    """
    Implements the git protocol. A protocol that simply proxies data
    without obfuscating them.
    """
    def __init__(self, isBridge):
        self.path = '/tmp/gittor'
        self.gitbuffer = GitBuffer(isBridge, self.path)

    def handshake(self, circuit):
        """
        The Circuit 'circuit' was completed, and this is a good time
        to do your transport-specific handshake on its downstream side.
        """
        # ensure there is only one thread running
        global gitPollingThreadLock
        global KeepGitPollingAlive
        KeepGitPollingAlive = False

        #wait until the while loop in the polling thread is done
        gitPollingThreadLock.acquire()
        KeepGitPollingAlive = True

        # start polling in the background
        thread.start_new_thread(self.__thread_gitPolling, (circuit,))
```

```
  def __thread_gitPolling(self, circuit):
    global gitPollingThreadLock
    global KeepGitPollingAlive
    try:
      while KeepGitPollingAlive and not circuit.closed:
        stack = self.gitbuffer.readStack()
        while stack:
          reactor.callFromThread(circuit.upstream.write, stack.pop())
      self.gitbuffer.cleanup()
    except Exception as e:
      log.debug("__thread_git_polling Exception: %s" % (str(e)))
    finally:
      gitPollingThreadLock.release()

  def receivedDownstream(self, data, circuit):
    """
    Got data from downstream; relay them upstream.
    """
    circuit.upstream.write(data.read())

  def receivedUpstream(self, data, circuit):
    """
    Got data from upstream; relay them downstream.
    """
    self.gitbuffer.write(data.read())

class GitClient(GitTransport):
  """
  GitClient is a client for the 'git' protocol.
  """
  def __init__(self):
    super(GitClient, self).__init__(False)

class GitServer(GitTransport):
  """
  GitServer is a server for the 'git' protocol.
  """
  def __init__(self):
    super(GitServer, self).__init__(True)
```

C.2. Helper code for encoding/decoding TCP streams in a Git repository

```
import pygit2, threading
import obfsproxy.common.log as logging

log = logging.get_obfslogger()

class GitBuffer:
  sequenceNumber = 0
  lock = threading.Lock()

  def __init__(self, isBridge, path):
    GitBuffer.lock.acquire()

    #The support for multiple simultaneous TCP connections is not
    #completed. Currently each branch is assigned a sequence number
    #which needs to be in sync between client and bridge but future
    #versions should use a unique identifier.
    GitBuffer.sequenceNumber += 1

    #constants
    self.PAYLOAD_FILENAME = 'payload'

    #init repository
    self.repo = pygit2.init_repository(path, bare=True)
    self.author = pygit2.Signature('test', 'test@example.com')

    #Initialize the bridge and client queues with an empty payload
    oid  = self.repo.create_blob('')
    bld = self.repo.TreeBuilder()
    bld.insert(self.PAYLOAD_FILENAME, oid, pygit2.GIT_FILEMODE_BLOB)
    treeid = bld.write()

    #Configure variables for referencing read and write queues
    if isBridge:
      log.debug("GitBuffer: Running as Bridge.")
      self.readRefName  = "refs/remotes/origin/fromClient" + \
        str(GitBuffer.sequenceNumber)
      self.writeRefName = "refs/heads/fromBridge" + \
        str(GitBuffer.sequenceNumber)
    else: #isClient
      log.debug("GitBuffer: Running as Client.")
      self.readRefName  = "refs/remotes/origin/fromBridge" + \
        str(GitBuffer.sequenceNumber)
```

```
      self.writeRefName = "refs/heads/fromClient" + \
        str(GitBuffer.sequenceNumber)

    #The last read commit from the read queue.
    #Initialize as None because branch hasn't been created
    self.readStackHead = None

    #Create and empty commit on write queue. This is required
    #because of how readStackHead is re-initialized.
    self.repo.create_commit(self.writeRefName,
      self.author, self.author, "msg", treeid, [])

    GitBuffer.lock.release()

  def cleanup(self):
    wref = self.repo.lookup_reference(self.writeRefName)
    wref.delete()
    try:
      rref = self.repo.lookup_reference(self.readRefName)
      rref.delete()
    except:
      pass # bridge did not setup its ref

  def write(self, data):
    #object
    oid  = self.repo.create_blob(data)

    #tree
    bld = self.repo.TreeBuilder()
    bld.insert(self.PAYLOAD_FILENAME, oid, pygit2.GIT_FILEMODE_BLOB)
    treeid = bld.write()

    #commit
    parent = [self.repo.lookup_reference(
      self.writeRefName).resolve().target.hex]
    self.repo.create_commit(self.writeRefName,
      self.author, self.author, "msg", treeid, parent)

  def readStack(self):
    stack = []
    #re-initialize if it hasn't been done before.
    if not self.readStackHead:
      try:
        newest = self.repo.lookup_reference(
```

```
        self.readRefName).resolve().target.hex
      for commit in self.repo.walk(newest, pygit2.GIT_SORT_NONE):
        oldest = commit.hex
      self.readStackHead = oldest
      log.debug("GitBuffer: re-initialize readStackHead %s" \
        % (self.readStackHead))
    except:
      pass # no ref with name readRefName -- yet
  if self.readStackHead:
    readRefHead = self.repo.lookup_reference(
      self.readRefName).resolve().target.hex
    for commit in self.repo.walk(readRefHead, pygit2.GIT_SORT_NONE):
      if commit.hex == self.readStackHead:
        break
      payload = self.repo[commit.tree[
        self.PAYLOAD_FILENAME].oid].read_raw()
      if len(payload) > 0:
        stack.append(bytes(self.repo[commit.tree[
          self.PAYLOAD_FILENAME].oid].read_raw()))
        log.debug("GitBuffer: adding %s to stack" % (commit.hex))
    self.readStackHead = readRefHead
  return stack
```