TOR - Didactic Pluggable Transport

Ioana-Cristina Panait[2(✉)], Cristian Pop[2], Alexandru Sirbu[2], Adelina Vidovici[2], and Emil Simion[1]

[1] Faculty of Applied Sciences, University Politehnica of Bucharest, Bucharest, Romania
esimion@upb.ro, esimion@fmi.unibuc.ro
[2] Faculty of Automatic Control and Computers, University Politehnica of Bucharest, Bucharest, Romania
{ioana.panait,cristian.pop,alexandru.sirbu,adelina.vidovici}@cti.pub.ro

Abstract. Considering that access to information is one of the most important aspects of modern society, the actions of certain governments or internet providers to control or, even worse, deny access for their citizens/users to selected data sources has lead to the implementation of new communication protocols. TOR is such a protocol, in which the path between the original source and destination is randomly generated using a network of globally connected routers and, by doing so, the client is not identified as actually accessing the resource. However, if the ISP knows that the first hop is part of TOR or if it can identify the contents of the exchanged packages as being TOR packages, by using advanced detection algorithms, it can still perform it's denial policies. These types of detection are circumvented by the usage of bridges (TOR routers which aren't publicly known) and pluggable transports (content changing protocols, in order to pass through as innocent-looking traffic). The development of a didactic pluggable transport in a simulated TOR network is the main purpose of this paper, in order to investigate the current state of the art of TOR development and analysis.

Keywords: TOR · Pluggable transport · ExperimenTOR · Obfsproxy

1 Introduction

This paper starts by presenting the motivation to develop a didactic pluggable transport and, also, some aspects of the TOR, such as its history, protocol, known vulnerabilities and some improvements, then inspects the current state of the art in terms of pluggable transports for TOR, followed by the main contribution of the article in our own implementation of a pluggable protocol over the simulated TOR network and finishing with the results of running TOR with the implemented protocol.

Many of the previous solutions based themselves on transforming the traffic between the source and the first hop have increased the amount of data sent by adding the overhead of masking the content, our proposed solution performs

© Springer International Publishing AG 2016
I. Bica and R. Reyhanitabar (Eds.): SECITC 2016, LNCS 10006, pp. 225–239, 2016.
DOI: 10.1007/978-3-319-47238-6_16

changes on the actual content in order to pass it as uncorrelated bytes which cannot be used in order to obtain information from the sent packets.

Thus, we decided to perform an inversion of the bit values in each of the content bytes of each packet, rendering the content unreadable without performing another inversion on the whole content. Knowing that both the sender and the first hop know of the usage of this pluggable transport, the data can be exchanged between them without a deep packet inspection determining that the traffic is part of the TOR network.

The main result of our work is the fact that the communication between the client and the TOR network, inside the isolated environment, works with our implemented pluggable transport, as well as with the original communication protocol and with only using bridges (without pluggable transport).

The results show us that, by using a bridge (the same first hop for all requests), the performance is slightly worse than the one when using the directory service to generate routes, as by changing the first hop we can get a better route and get better performances. Further comparing the results from the bridge tests, with and without pluggable transports, we see that using the pluggable transport comes with a small increase in duration, accounting for the coding and decoding of the content.

2 Motivation

In a world in which global communication is considered as one of the modern building blocks of modern civilization, the Internet, which appeared in the early 1990s, has been a major influence in the way information is exchanged between point A and B, allowing the interconnection of computers from all around the world. However, as with anything man made, malicious uses of this can produce data leaks, causing major problems to all the parties involved, even without their knowledge.

Thus, the protection of data transmission is one of the major concerns when talking about information exchange and, because of this, many protocols have been invented and implemented in order to allow the secure transfer of information from any sources. Many of these are at an application level, meaning that the data is encrypted at the source and decrypted at the destination. The data flow, however, is usually the same and a man in the middle attack, with sufficient knowledge, can disrupt the transmission and track the sender and receiver and, given sufficient time, can try and break the protocol of their transfer or at least can trace the pattern of communication and can cause harm to one of the entities involved by attacking the other.

This vulnerability of data transmission, that the communication can be traced back to the source and destination, is also important when talking about security. This is where the TOR protocol tries to come with a solution, in which the exchange between the two is anonymous, using a global infrastructure of servers. The route used by the sender is chosen at the beginning of the transfer and is changed at regular intervals, in order to not permit the analysis of traffic,

and also almost no servers know their role and the route before or after them besides their neighbors (only the exit node knows that the next destination is the original destination the sender wanted to contact), allowing for the actual sender to be forgotten when the data arrives at its final receiver.

However, many internet service providers try (or are obliged by the law) to not allow the use of TOR. The most basic way in which this is done is by blacklisting the public IPs of known TOR servers, but this is countered by the usage of bridge relay servers, which aren't listed anywhere and which allow connection to the network. A more intrusive way is to do deep packet inspection, in which the actual data is inspected and, from known patterns, it can be determined that it uses TOR and, thus, can deny the sending of the packet. For this issue, pluggable transports have been introduced to TOR, in which the traffic between the client and bridge is transformed into innocent-looking traffic instead of the normal TOR flow, tricking the DPI into allowing the packages.

3 TOR

3.1 Protocol

TOR is the second generation of Onion Routing and the name of an Internet network that allow people to communicate anonymously. Onion Routing is a distributed overlay network designed to anonymise applications like web browsing, instant messaging or secure shell, TCP-based applications by encryption in the application layer of a communication protocol stack. Clients choose a path through this network by building a circuit made of nodes. Each node/onion router knows only its predecessor and its successor [2].

To hide the identity over the Internet, TOR uses a group of volunteer-operated servers/relays which are employed by its users by connecting through a series of virtual tunnels. TOR encrypts the information several times and sends it through the circuit. The IP address of the destination is also encrypted. Each relay decrypts only a layer of encryption to reveal the necessary information about the next node in the circuit as we can see in Fig. 1.

The TOR network can be used to transport TCP streams anonymously. The network is composed of a set of nodes that act as relays for a number of communication streams, from different users. Each TOR node tries to ensure that the correspondence between incoming data streams and outgoing data streams is obscured from the attacker. Therefore the attacker cannot be sure about which of the originating user streams corresponds to an observed output of the network.

Each onion router maintains a long-term identity key and a short-term onion key. The identity key is used to sign TLS certificates, to sign the onion router's descriptor (a summary of its keys, address, bandwidth, exit policy, and so on), and (by directory servers) to sign directories. The onion key is used to decrypt requests from users to set up a circuit and negotiate ephemeral keys. The TLS protocol also establishes a short-term link key when communicating between onion routers. Short-term keys are rotated periodically and independently, to limit the impact of key compromise.

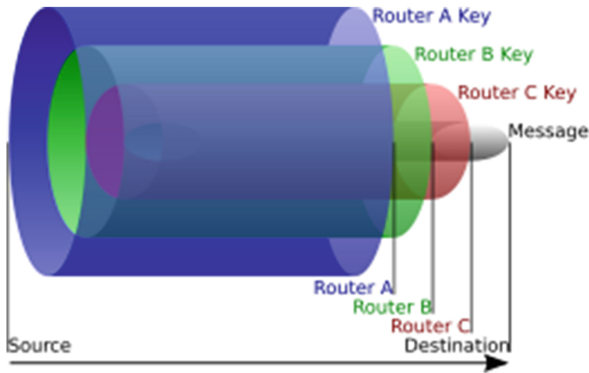

Fig. 1. Onion routing (Picture from Security Stack Exchange http://security.stackexchange.com/questions/76438/about-onion-packet-and-onion-routing).

Onion routers communicate with one another, and with users' of onion proxys, via TLS connections with ephemeral keys. Using TLS conceals the data on the connection with perfect forward secrecy, and prevents an attacker from modifying data on the wire or impersonating an onion router.

The TOR architecture is similar to conventional circuit switched networks. The connection establishment has been carefully crafted to preserve anonymity, by not allowing observers to cryptographically link or trace the route that the connection is using. The initiator of the stream creates a circuit by first connecting to a randomly selected TOR node, negotiating secret keys and establishes a secure channel with it. The key establishment uses self-signed ephemeral Diffie-Hellman key exchange and standard Transport Layer Security (TLS) is further used to protect the connections between nodes and provide forward secrecy.

All communications are then tunneled through this circuit, and the initiator can connect to further TOR nodes, exchange keys and protect the communication through multiple layers of encryption. Each layer is decoded by a TOR node and the data is forwarded to the next Onion router using standard route labeling techniques.

Finally, after a number of TOR nodes are relaying the circuit (by default three), the initiator can ask the last TOR node on the path to connect to a particular TCP port at a remote IP address or domain name. Application layer data, such as HTTP requests or SSH sessions, can then be passed along the circuit as usual (Fig. 2).

TCP streams traveling through TOR are divided and packaged into cells. Each cell is 512 bytes long, but to cut down on latency it can contain a shorter useful payload. This is particularly important for supporting interactive protocols, such as SSH, that send very small keystroke messages through the network. TOR does not perform any explicit mixing. Cells are stored in separate buffers for each stream, and are output in a round-robin fashion, going round the

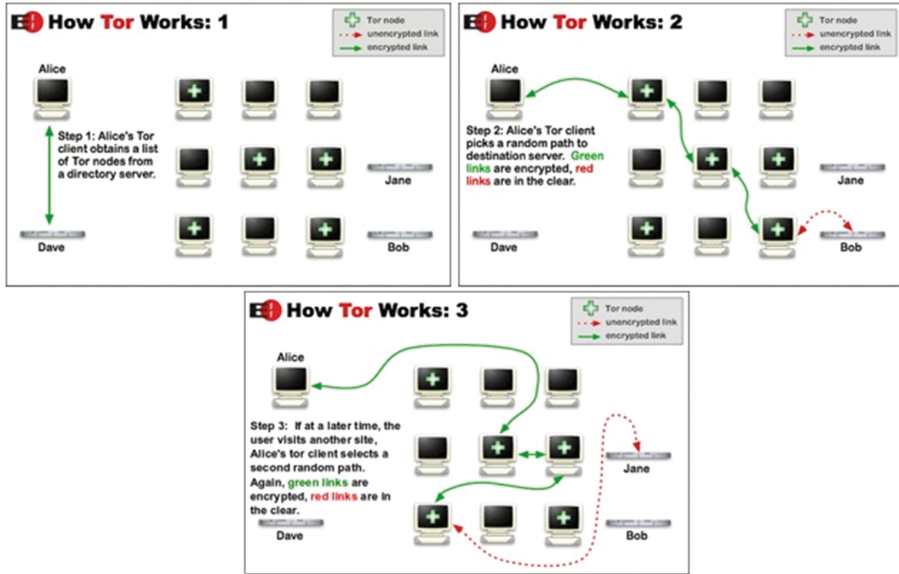

Fig. 2. TOR protocol (Picture from TOR Project Overview https://www.torproject.org/about/overview.html.en#thesolution).

connection buffers. This ensures that all connections are relayed fairly, and is a common strategy for providing best effort service.

Importantly, when a connection buffer is empty, it is skipped, and a cell from the next non-empty connection buffer is sent as expected. Since one of the objectives of TOR is to provide low latency communications, cells are not explicitly delayed, reordered, batched or dropped, beyond the simple-minded strategy described above.

TOR has some provisions for fairness, rate limiting and to avoid traffic congestion at particular nodes. Firstly, TOR implements a so-called token bucket strategy to make sure that long-term traffic volumes are kept below a specified limit set by each TOR node operator. Since the current deployment model relies on volunteer operators, this was considered important.

This approach would not prevent spikes of traffic from being sent, and propagating through a connection. These spikes of data would, of course, be subject to the maximum bandwidth of each node, and could saturate the network connection of some TOR nodes.

To avoid such congestion, a second mechanism is implemented. Each stream has two windows associated with it, the first describes how many cells are to be received by the initiator, while the other describes how many are allowed to be sent out to the network. If too many cells are in transit through the network and have not already been accepted by the final destination the TOR node stops accepting any further cells until the congestion is eased.

It is important to note that this mechanism ensures that the sender does not send more than the receiver is ready to accept, thereby overfilling the buffers at intermediary TOR nodes. It also makes sure that each connection can only have a certain number of cells in the network without acknowledgment, thus preventing hosts from flooding the network. TOR does not, however, artificially limit the rate of cells flowing in any other way [4].

Each TOR circuit can be used to relay many TCP streams, all originating from the same initiator. This is a useful feature to support protocols such as HTTP, that might need many connections, even to different network nodes, as part of a single transaction.

Unused TOR circuits are short-lived replacements are set up every few minutes. This involves picking a new route through the TOR network, performing the key exchanges and setting up the encrypted tunnels [3].

3.2 Known Vulnerabilities

Client can obtain all TOR routers information. In the process of circuit establishment, each TOR client fetches all onion routers information from Directory Server, which gives an adversary the ability to obtain a total TOR network view. With the complete network view it is possible for the adversary to perform DDOS attack or low-cost traffic attack on TOR network.

TOR does not use any batching strategy. To decrease the latency of communication, TOR does not consider any batching strategy in node design. Instead cells from different circuits are sent out in a round robin fashion. When a circuit has no cells available, it is skipped and the next circuit with cells waiting to be delivered is handled. This means that the load on the TOR node affects the latency of all connection circuits switched through this node. An extra connection can result in higher latency of all other connections routed through the same TOR node. So by producing specific traffic, and measuring the latency of all TOR nodes, the adversary can identify all relay nodes of target circuit.

TOR does not check TOR node information. Within TOR's routing model, each TOR node advertises its information such as uptime, IP address, bandwidth and so on in Directory Server. Directory Server does not perform any checking on the information. OP chooses a relay node to establish the circuit according to the information registered in Directory Server. It is possible for the adversary to perform low-resource routing attack with this weakness because an adversary can use the weakness to advertise very high bandwidth, very long uptime and unrestricted exit policies.

The information is reported by TOR node voluntarily. When the TOR node exits TOR network, it is possible for the node not to report its withdraw. In such case both Directory server and other TOR nodes would not know the situation. It causes the OP failure when relaying cells along the circuit passing through the node or trying to establish the circuit with the node [1].

3.3 Improving Performance

It can be seen that, despite previous research proposals, scalability problems are still lurking in the future of TOR. P2P proposals can not be adopted because their lookup process reveals circuit information, and they are susceptible to attacks where the adversary controls a large fraction of the network by introducing bogus nodes (using a botnet, for example).

PIR-Private Information Retrieval approaches look promising, but they still need further investigation. PIR-TOR, for example, requires node reuse in its CPIR (Single-server computational PIR schemes) instantiation, lowering the security of TOR, while in its IT-PIR (Information-theoretic PIR schemes) instantiation, requires multiple guards for each user to act as PIR servers [6].

This creates tension with recent considerations to reduce the number of guards to improve anonymity. Providing incentives for users to run as routers can have a positive impact on scalability and congestion. Incentive-based proposals suffer from shortcomings that need to be addressed.

One promising direction is an approach based on proof-of-bandwidth like tor-coin, where routers are rewarded with digital coins based on how much bandwidth they use relaying anonymous traffic. One challenge for a proof-of-bandwidth protocol is performing secure bandwidth measurements to ensure all network participants can easily verify that routers indeed spend what they claim to spend [7].

Furthermore, while there have been several transport layer proposals that aim to reduce congestion in TOR, it is still unclear what transport design provides the required trade-off between anonymity and performance for TOR. There is a need to experimentally compare the different transports under realistic user, network and traffic models that can emulate the real TOR network. Once a transport design is identified, a deployment plan must be carefully crafted in order to gradually and smoothly upgrade the network without denying service to its users [5].

4 Pluggable Transports

In order to restrain the Internet access when using TOR, some countries or ISPs use different techniques for detecting unwanted Internet traffic flows by protocol. If the ISP is filtering connections to TOR relays, there is a solution for overpassing this issue by using bridge relays (or bridges). These are also TOR relays, but they are not listed in TOR directory and there is no complete public list for them. Besides filtering connections, ISPs can also analyze the traffic by using DPI (Deep Packet Inspection), so the censor will be able to recognize and filter TOR traffic based on some samples. A solution for this problem is given by the use of pluggable transports.

Pluggable transports can transform the data passing between the client and the bridge so that it looks like "normal/expected traffic". This way, the censors cannot detect and filter TOR traffic as long as they cannot decide if a TOR connection is in use.

However, we cannot state that pluggable transports are undetectable. Given enough time for research into how these methods manipulate traffic, one can find means to detect when certain pluggable transports are used. This way, some transports become deprecated over time and they need to be replaced by more improved ones.

As state of the art, there are several pluggable transports already deployed and also there are several in progress to be deployed or developed. Obfsproxy is a framework used for implementing new pluggable transports and it is written in Python. It is an application independent from TOR which has a client and a server that support numerous pluggable transports protocols. The obfsproxy client is placed between TOR client and the censor and the obfsproxy server is placed between the censor and TOR bridge, as we can see in Fig. 3. Some of the pluggable transports supported are obfs2 and obfs3 (protocol obfuscation layer for TCP protocols). Flashproxy brings another overview of skipping censors' system and allow access to TOR [9]. It is a proxy that runs in a web browser and checks for clients that request access, then it transmits data between those clients and the TOR relay. The technologies used in implementing Flashproxy are JavaScript and WebSocket, and the objective of this project is to outrun the censors' ability to recognize the bridge's IP address, by creating many temporary bridge IP addresses.

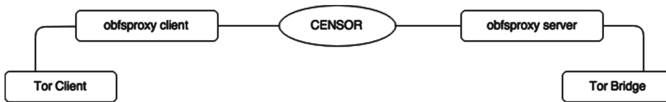

Fig. 3. Obfsproxy

Another deployed transport is Format-Transforming Encryption (FTE) [8] which modifies TOR traffic to streams that match a user-specified regular expression. FTE is a novel cryptographic primitive, which differs from a traditional one by the introduction of a new input as a set descriptor. In the traditional form, the cryptographic primitive has a key and message as input and outputs a simple ciphertext based on them. FTE has a key, a message and a format as input and outputs a ciphertext in the format set described. This way, censored traffic can pass as legitimate traffic, because of its resemblance with normal traffic, like HTTP for instance.

Another pluggable transport which is part of Obfsproxy framework previously describer is ScrambleSuit [10]. The exchanged traffic between the TOR client and the TOR bridge is encrypted, authenticated and disguised. From a technical point of view, this protocol protects against active probing attacks and can generate unique flow signature by altering the inter-arrival time and the packet length distribution. As an observation, ScrambleSuit can transport many other protocols besides TOR, like VPN, SSH etc.

Meek is a transport used to relay traffic through a third-party server like a CDN, which is hard to block by the censor. The method is called "domain fronting", which means that different domain names are used for different communication layers. The request of the meek-client has the domain that appears on the "outside" of the request, and a different domain that appears on the "inside" of the request, and cannot be seen by the censor. The CDN does see the inside domain and forwards the packet accordingly to a meek-server from a TOR bridge. The meek-server will process the data and send it to TOR.

Obfs4 is a transport which resembles ScrambleSuit, but has a different public key obfuscation technique and a protocol for one-way authentication. The project is written in Go and it is faster than ScrambleSuit.

Obfsclient is a pluggable transport proxy, which is written in C++ and implements the client side of obfs2, obfs3, ScrambleSuit.

SkypeMorph currently has an undeployed status and it is designed to cover TOR traffic flows by using a widely known protocol over the Internet. The target protocol investigated is Skype video call [11].

These are just a few of all the pluggable transports, implemented or in progress so far, and they can be found on the official page of TOR project [12]. The objective is to have as many designs as possible in order to better avoid capturing the TOR traffic by deep packet inspection.

5 Architecture and Implementation

As TOR is a fully functional protocol, already running over the Internet, the addition of a new pluggable transport requires, therefore, its development to be done in an isolated environment, in order to not add new routers with functionality which may negatively influence the activity of clients which already use the service.

5.1 Development Environment - ExperimenTOR

Thus, in order to start our implementation we needed to create an environment in which to run our development and testing process. The environment needed to actually run TOR code and not simulate the packages sent between the entities (as the pluggable transport needs to actually send and receive packages over the network), to be easy to start, modify and analyze (in order to be able to perform multiple tests on possible different networks) and to be reliable (elements must not break during usage).

These requirements meant that the best option would be to use an already existing tool. The TOR project presents two options in this matter: the Shadow simulator (which has an implemented extension for TOR) and the experimenTOR simulator, presented as an testbed for TOR development. As the second one is solely oriented on TOR simulation, we decided to utilize it as our environment.

However, as experimenTOR is an old tool and its released version dated back from 2011, we encountered several problems during its setup and configuration, presented below as well as our solutions for each one of them:

- the solution came as a bundle of two virtual machines, one containing the ModelNet simulated network and one containing the actual running code; the latter was installed on an Ubuntu 11.04 machine, which finished it's support life and this meant that we needed to change it's rpm sources in order to use the archived latest versions
- in order to work, TOR routers require signed certificates, to identify themselves in the network to the other entities; as the virtual machines were from 2011, the allocated certificates were expired and, thus, when running TOR, the routers would stop working, requiring correct certificates; our solution was rather hackish, but worked in the environment - we turned the clock back for the virtual machine in 2011, re-activating the allocated certificates
- the TOR code provided was at version 0.2.3.0, largely outdated from the latest version of 0.2.7.6; it also didn't have support for bridges or pluggable transports, meaning that we needed to update to a newer version in order to be able to do our intended work over the network
- version 0.2.7.6 of TOR requires the minimum version 1.01h for OpenSSL; the latest version in the rpm sources was 0.99o, meaning that we needed to install OpenSSL from sources which usually has a degree of danger and may cause incompatibilities with already generated elements without any further issues
- the configuration files for TOR routers and clients changed from the format present in the tool in 2011, so we needed to bring them up to date
- manually install obfsproxy, as it wasn't already provided

By doing the previous changes, we managed to create a working environment with 10 routers, with the latest versions for all the needed tools (TOR and obfsproxy), in which to do our research.

5.2 Obfsproxy

The simplest way to implement a new pluggable transport was to use the obfsproxy. The framework comes with a list of already implemented pluggable transports as presented beforehand, but can also permit the implementation of new ones easily. The framework takes care of the full pluggable transport API implementation and network communication, leaving to developers only the implementation of the content changing algorithm.

In order to utilize pluggable transports, the TOR clients and servers need to be configured to use obfsproxy. The client needs to be informed that it needs to use bridges (thus, it will choose the first hop from the list of provided bridges in the configuration file) and, further, to use the named transport (in our case, reverse) which is provided by obfsproxy. The managed parameter sent to obfsproxy states that the connection between client and proxy is fully managed by the TOR client. The server is configured in order to run as a bridge relay,

listening for content changed with the named transport (the same one used as the client, reverse), again by using obfsproxy in a managed state.

```
Client configuration
UseBridges 1
ClientTransportPlugin reverse exec obfsproxy managed
Bridge reverse 127.0.0.1:39201

Server configuration
BridgeRelay 1
ServerTransportPlugin reverse exec obfsproxy managed
ServerTransportListenAddr reverse 127.0.0.1:39201
```

5.3 Proposed Solution Pluggable Transport Algorithm

As almost all of the previous solutions based themselves on transforming the traffic between the source and the first hop, increasing the amount of data sent by adding the overhead of masking the content, our proposed solution goes a different path, by performing changes on the actual content in order to pass it as uncorrelated bytes which cannot be used (without other changes) in order to obtain information from the sent packets.

In order to perform a proof of concept of this concept, we decided to implement the simplest of changes, in order to allow the masking of content. Thus, we decided to perform an inversion of the bit values in each of the content bytes of each packet, rendering the content unreadable without performing another inversion on the whole content. Knowing that both the sender and the first hop know of the usage of this pluggable transport, the data can be exchanged between them without a deep packet inspection determining that the traffic is part of the TOR network.

This proposed solution comes with the following benefits:

- no overhead over the original content, as each of the bytes of the original content gets changed to another byte of data
- easy and fast operation in order to encode/decode the content, without a big impact on the transmission speed
- the simple change drastically changes the semantics of the content, allowing it to pass through filters which only check the content

However, as the change is simple, it can also be added to the deep packet inspection solutions in order to detect traffic which uses this change. In this case, the time needed to inspect one packet will increase at least twofold, as the original packet needs to be inspected first, then the packet needs to be transformed and the checked again, a time increase that isn't feasible when inspecting packages on the go without impacting the client performance. This can also be increased if a more complex algorithm is used on the content, as this inversion is just a proof of concept that such a pluggable transport can be implemented.

The implementation of this algorithm as part of the obfsproxy came as an extension to the BaseTransport protocol. As the algorithm is symmetric (the client and server do the same operation), the difference between the server and client functions is strictly concerning the flow of data. Thus, the client will receive the data from downstream, change it and send it upstream and the server receives the data from upstream, changes it again to get the original data and then sends it downstream, in order to be actually used. The added class is the following:

```
class ReverseTransport(BaseTransport):
    """
    Implements the reverse protocol. A protocol that reverses bytes
    and then proxies data.
    """
    def __init__(self):
        """
        If you override __init__, you ought to call the super method too.
        """
        super(ReverseTransport, self).__init__()
    def receivedDownstream(self, data):
        """
        Got data from downstream; reverse and relay them upstream.
        """
        buffered = data.read()
        reverse=''
        for i in range(0,len(buffered)):
              reverse+=chr(~ord(buffered[i]) & 0xFF)
        self.circuit.upstream.write(reverse)
    def receivedUpstream(self, data):
        """
        Got data from upstream; reverse and relay them downstream.
        """
        buffered = data.read()
        reverse=''
        for i in range(0,len(buffered)):
              reverse+=chr(~ord(buffered[i]) & 0xFF)
        self.circuit.downstream.write(reverse)

class ReverseClient(ReverseTransport):
    """
    ReverseClient is a client for the 'reverse' protocol.
    Since this protocol is so simple, the client and the server
    are identical and both just trivially subclass ReverseTransport.
    """

class ReverseServer(ReverseTransport):
    """
    ReverseServer is a server for the 'reverse' protocol.
    Since this protocol is so simple, the client and the server
    are identical and both just trivially subclass ReverseTransport.
    """
```

As obfsproxy is written in python, reversing the bits of a byte needed to also be implemented in the same language. By doing $chr(\sim ord(byte)\&0xFF)$, this functionality is achieved (the $\sim$ operator inverts the bits of an integer number). The usage of the extra $\&0xFF$ was mandatory, as the $\sim$ operator returns a signed number and *chr* needs a value between 0 and 255 in order to work.

6 Results

The main result of our work is the fact that the communication between the client and the TOR network, inside the isolated environment, works with our implemented pluggable transport, as well as with the original communication protocol and with only using bridges (without pluggable transport).

Having the possibility of running the client with any of these communication protocols, we decided to run a test in order to determine the possible performance differences of the three. Thus, we booted up the network with 10 routers with the following flags:

– Router 1 - Exit Fast HSDir Running Stable V2Dir Valid
– Router 2 - Fast Running V2Dir Valid
– Router 3 - Exit Fast Running V2Dir Valid
– Router 4 - Fast Guard HSDir Running Stable Valid
– Router 5 - Fast Guard HSDir Running Stable V2Dir Valid
– Router 6 - Fast Running Stable V2Dir Valid
– Router 7 - Fast Guard HSDir Running Stable Valid
– Router 8 - Fast Running Valid
– Router 9 - Fast HSDir Running Stable Valid

The first five nodes were also directory services and the bridge service ran on router 6 (for the two tests requiring bridges). After the network started, consensus was reached and all routers were connected, the client connects to the network using one of the three connection possibilities and, then, the connection is used to download files from a webserver. By varying the sizes of the files, the performances for each of the connection method can be obtained. For each file size, we performed 10 tests and the given value is the mean value of all.

The results show us that, by using a bridge (the same first hop for all requests), the performance is slightly worse than the one when using the directory service to generate routes, as by changing the first hop we can get a better route and get better performances.

Further comparing the results from the bridge tests, with and without pluggable transports, we see that using the pluggable transport comes with a small increase in duration, accounting for the coding and decoding of the content. The difference, however, is small when compared to the first test, showing that, by having an overhead of around 10–15 %, we can achieve a better bypassing of deep packet inspectors (Table 1).

Table 1. Results

	100 kb	200 kb	300 kb	500 kb	5 mb
Directory connection	0.05 s	0.06 s	0.07 s	0.1 s	1.3 s
Bridge (no pluggable transport)	0.05 s	0.07 s	0.07 s	0.1 s	1.4 s
Bridge with reverse transport	0.05 s	0.07 s	0.08 s	0.12 s	1.47 s

7 Conclusion

The introduction of new elements in TOR, such as bridges and pluggable transports, has permitted more and more users to bypass security measures and access information denied to them until now, due to the inspection systems put into places by governments and ISPs.

The development of such a pluggable transport requires the existence of an isolated environment, in order not to interfere with the actual usage of the TOR network. The existing tools for such an environment are outdated, but with some changes, it can be brought up to date in order to implement the most recent version of TOR and obfsproxy, in order to properly simulate real-life conditions. By using obfsproxy, the addition of a new pluggable transport is facilitated, as the developer is left to implement data and decoding, leaving the framework to do the actual communication.

The results of our tests show that using these censorship circumventing methods adds a slight overhead over the traditional way of using the TOR network. However, the overhead is more than manageable as these methods are used when access is more important than speed. In the future, we wish to implement a more complex algorithm for data coding and decoding (as the one we chose here was for the sake of having a proof of concept of using the environment) and to run tests on different type of networks, not only with the default one.

Acknowledgments. This work partially supported by the Romanian National Authority for Scientific Research (CNCSUEFISCDI) under the project PN-II-PT-PCCA-2013-4-1651.

References

1. Xin, L., Neng, W.: Design improvement for TOR against low-cost traffic attack and low-resource routing attack privacy enhancing technologies. In: International Conference on Communications and Mobile Computing (2009)
2. Dingledine, R., Mathewson, N., Syverson, P.: TOR: the second-generation onion router. Information Security Research Group (2014)
3. Murdoch, S.J., Danezis, G.: Low-cost traffic analysis of TOR. In: IEEE Symposium on Security and Privacy (2005)
4. Murdoch, S.J.: Covert channel vulnerabilities in anonymity systems. Technical report (2007)

5. AlSabah, M., Goldberg, I.: Performance and security improvements for TOR: a survey. In: International Association for Cryptologic Research (2015)
6. Mittal, P., Olumofin, F.: PIR-TOR: scalable anonymous communication using private information retrieval. USENIX Security (2014)
7. Ghosh, M., Richardson, M.: A TorPath to TorCoin: proof-of-bandwidth altcoins for compensating relays. USENIX Security (2011)
8. Dyer, K.P, Coull, S.E., Ristenpart, T., Shrimpton, T.: Protocol misidentification made easy with format-transforming encryption. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 61–72. ACM (2013)
9. Fifield, D., Hardison, N., Ellithorpe, J., Stark, E., Boneh, D., Dingledine, R., Porras, P.: Evading censorship with browser-based proxies. In: Fischer-Hübner, S., Wright, M. (eds.) PETS 2012. LNCS, vol. 7384, pp. 239–258. Springer, Heidelberg (2012)
10. Winter, P., Pulls, T., Fuss, J.: ScrambleSuit: a polymorphic network protocol to circumvent censorship. In: Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society, pp. 213–224. ACM (2013)
11. Mohajeri Moghaddam, M., Li, B., Derakhshani, M., Goldberg, I.S.: protocol obfuscation for TOR bridges. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 97–108. ACM (2012)
12. TOR Project - Pluggable Transports. https://www.torproject.org/docs/pluggable-transports.html.en