Port-Scanning Resistance in Tor Anonymity Network

Presented By: Shane Pope (Shane.M.Pope@gmail.com)

Dec 04, 2009

In partial fulfillment of the requirements for graduation with the
Dean's Scholars Honors Degree in Computer Science

Department of Computer Science
University of Texas at Austin

Thesis Advisor: Dr. Vitaly Shmatikov
Additional Help: Roger Dingledine

Abstract

The Onion Router (Tor) is an anonymity network that allows users to perform web browsing and other internet activities anonymously. A Tor user's traffic is routed securely through several other Tor relays before making its way to its destination. No one except the final Tor router in this circuit knows what the final destination of users traffic is and each router in the circuit only knows about the previous and next router. Tor users get the list of Tor IP addresses from a dedicated server which lists most of the Tor routers. With this list they can create random circuits through the internet to route their traffic. Governments that censor the internet with country-wide firewalls want to block Tor, because it allows users to circumvent the censorship. China has begun blocking Tor by downloading the list of all public Tor IP addresses and blocking them. [1] There are still options for internet users in China to access the Tor network. One option is unpublished Tor relays whose internet addresses are shared via email and instant messenger instead of in a public directory like normal Tor relays. Since these unpublished routers cannot be easily downloaded in bulk like the published Tor routers, detecting and blocking unpublished routers is the obvious next step for China and other censoring nations. Currently it is possible to detect these unpublished Tor relays by running Tor and attempting to connect to every internet address on ports Tor commonly runs. If a computer responds as Tor would, you know it is running Tor and can thus block the internet address. In this paper I present and implement a protocol which decreases the ease of detection of these unpublished relays, by hiding them behind a web server to prevent this type of scanning.

Introduction to Tor

The Onion Router (Tor) is an anonymity network which aims to prevent traffic analysis for its users. Traffic analysis is an attack in which the attacker looks at packet source and destination data to find out who is talking to whom. Tor prevents this is by encrypting data in layers of encryption, and sending the encrypted data through a random series of Tor relays hosted by volunteers around the world. [2]

Different kinds of people use the Tor network for different reasons. Military and Law enforcement use it for sting operations so their IP addresses cannot be traced to known

police or military addresses. Military field agents use it to prevent being tracked down for visiting military related websites while in the field. Some journalists use Tor to remain anonymous from governments who would shut them down. Tor is also commonly used by individuals to maintain privacy and evade harsh firewalls, especially firewalls with strict government enforced censorship. [2]

The basic Tor protocol starts with a Tor client requesting a list of Tor relays from a trusted Tor directory server. The Tor directory server returns a subset of public Tor relays to the client. When the client needs to build a connection, the client creates a chain of random relays from their list of relays. Next, the client begins negotiating keys with each one, through the chain. In doing this each relay only knows the previous and next node in the chain. The chain then ends at an exit relay which exchanges data with the client's actual destination.

Another method of access to the Tor network is through bridges. Bridges are Tor relays that are not published in the directory. The purpose of bridges is to allow people to access the Tor network even if all public relays become blocked. Bridges are shared through several methods including e-mail, instant messages, and web pages. The recent blocking of all public Tor relays and some bridges in China showed the importance of bridges as the estimated bridge use increased over 70 times to over 10,000 bridge users. [1]

The Problem and Attack Model

Chinese and various other state-run firewalls block their citizens from accessing websites by IP addresses and word filters. Because Tor prevents traffic analysis, as long as the end node is not behind a firewall, Tor users can circumvent government censorship. Therefore China and other censoring entities would like to block their internet users from connecting to Tor.

There are multiple methods for censoring entities to block their users from connecting to Tor. We will assume these censoring entities have both man-in-the-middle capabilities and a firewall that can block based on IP address. These assumptions are reasonable because, as in China, the Internet service providers are government-controlled and therefore the government can look at all packets and set rules to allow or deny anything through the country-wide firewall.

The first and simplest attack method for blocking user access to Tor is to download the

list of Tor relays from a public Tor directory. Since the list contains the IP addresses of all of the public Tor relays, the IP addresses can be added to the firewall block list. This methodology blocks a majority of the relays, since most relays are in the public directory. This form of blocking is already occurring inside of China. [1]

To get around this simple form of blocking, Tor uses unpublished bridges shared via email, instant messenger, and the web. Some of these bridges are listed on the web and can be easily downloaded and added to the firewall. But the ones shared over email and instant messenger between acquaintances still prevent easy downloading and blocking by the government.

While bridges are a good short term solution, an adversary can identify if a machine is running Tor by attempting to connect to ports on the machine and following Tor protocol. If any of the ports respond, following Tor protocol, we know they are running Tor and can therefore detect Tor bridges. This active scanning is the second attack method; it requires more computation and effort and is not currently being used by any governments.

With this current bridge protocol it is quite easy, with enough computational power and bandwidth, to detect bridges and add those IP addresses to the firewall as well. This is made easier since Tor runs commonly on port 443 to allow people behind port restrictive firewalls to reach it. Therefore someone can detect most bridges by only scanning this port reducing the scanning time extensively. The search time can be reduced further by only scanning IP blocks of cable modem and DSL networks, since most bridges are hosted on personal computers.

This second attack method is one obvious next step to further blocking Tor inside of China and other nations, especially since bridge use is on the rise. [1] The protocol presented below aims to prevent this attack method by inhibiting this kind of port-scanning, thus increasing the cost of detecting a Tor bridge.

A third attack method is to actively analyze traffic by timing, size and quantity. This kind of attack requires incredible resources and is prone to false positives unlike the others. Since Tor traffic is different than normal web traffic, one can imagine building a statistical model to differentiate Tor traffic from other kinds of traffic. Due to the computational power required for this attack, we do little to address this attack. However, our protocol does hide the initial TLS handshake behind a web-browser-identical TLS handshake, thus preventing an attacker from detecting a difference between handshakes and therefore detecting Tor

traffic.

Protocol Introduction and Design Objectives

The most important objective is to prevent detection of Tor bridges via port-scanning, and increase the complexity of detecting Tor bridges by any kind of scanning. To do this our protocol attempts to respond like a web server to any user who does not know some password. By doing this only a client who knows the password will get a Tor response, anyone else should get a web server response and not know that the computer is running a Tor bridge. The web server our protocol will attempt to replicate is the open source web server Apache.

Another objective is to make sure that no matter how a adversary knocks on the bridge's ports they will not be able to tell that they are running Tor. Therefore, our protocol must be bug-compatible with Apache. Otherwise, if an adversary could find a single way of knocking at Tor which gave a different response than Apache would give, the adversary would know the computer was running Tor. In order for the bridge to maintain compatibility and indistinguishability with Apache, our bridge will run Apache, instead of trying to emulate it.

Finally we want this protocol to work with existing Tor relays. Therefore the main Tor protocol remains exactly the same.

We solve these objectives by having the bridge host run both Apache and a Tor bridge. The Tor bridge only accepts connections from Apache. If a Tor client wants to use the bridge, they sneak the bridge's password in through a GET query string encrypted under SSL. If the password is correct, the Apache web server decrypts the new SSL layer and passes all future data to the local Tor bridge. Otherwise, if the password is incorrect, Apache will handle the request as it normally would if the Tor module was not there.

Introduction to Apache HTTP Server

Apache HTTP is one of the most widely used web servers. It is open-source and has a wide variety of modules supporting features such as SSL, proxying, and CGI. Apache is composed of a common core, a platform dependent layer, and the modules.

The modules handle content generation, and exactly one content generator must be run

for every HTTP request. Examples of content generators include HTTP handling, CGI handling, and proxy handling. Apache splits the request into different phases to decide how to handle it. For example, a user may need to be authorized before the content generator will handle the request. Modules have the ability to hook a handle into any of the phases on the processing axis in Figure 1. [4] If a filter decides to not handle the request, DECLINE or a HTTP error is returned. If DECLINE is returned then another content generator will take care of the request. Otherwise at the end of content generation OK is returned, notifying the server that it was handled.

Along with content generation, every request has an input filter chain and output filter chain. Input filters perform actions such as reading data from the socket, decrypting SSL, logging requests and more. Output filters perform actions like encrypting SSL responses, logging, and writing data to the connection's socket.

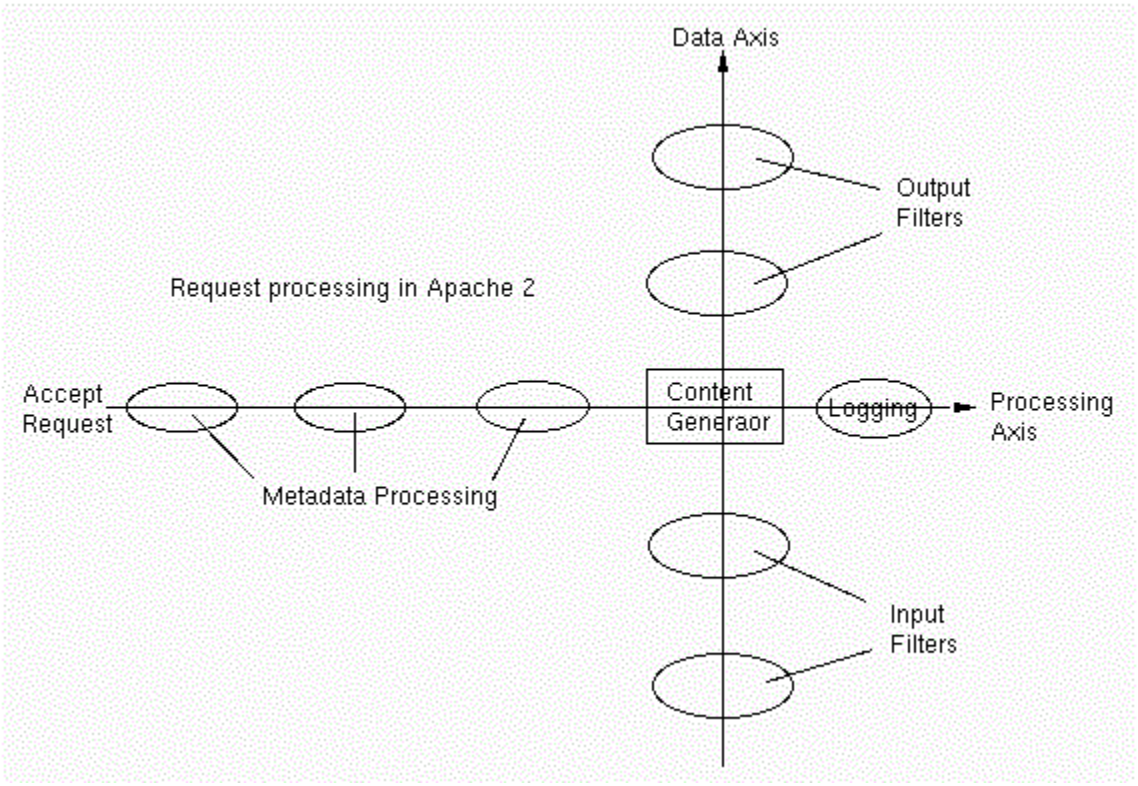

Figure 1: Request Processing in Apache [4]

Protocol

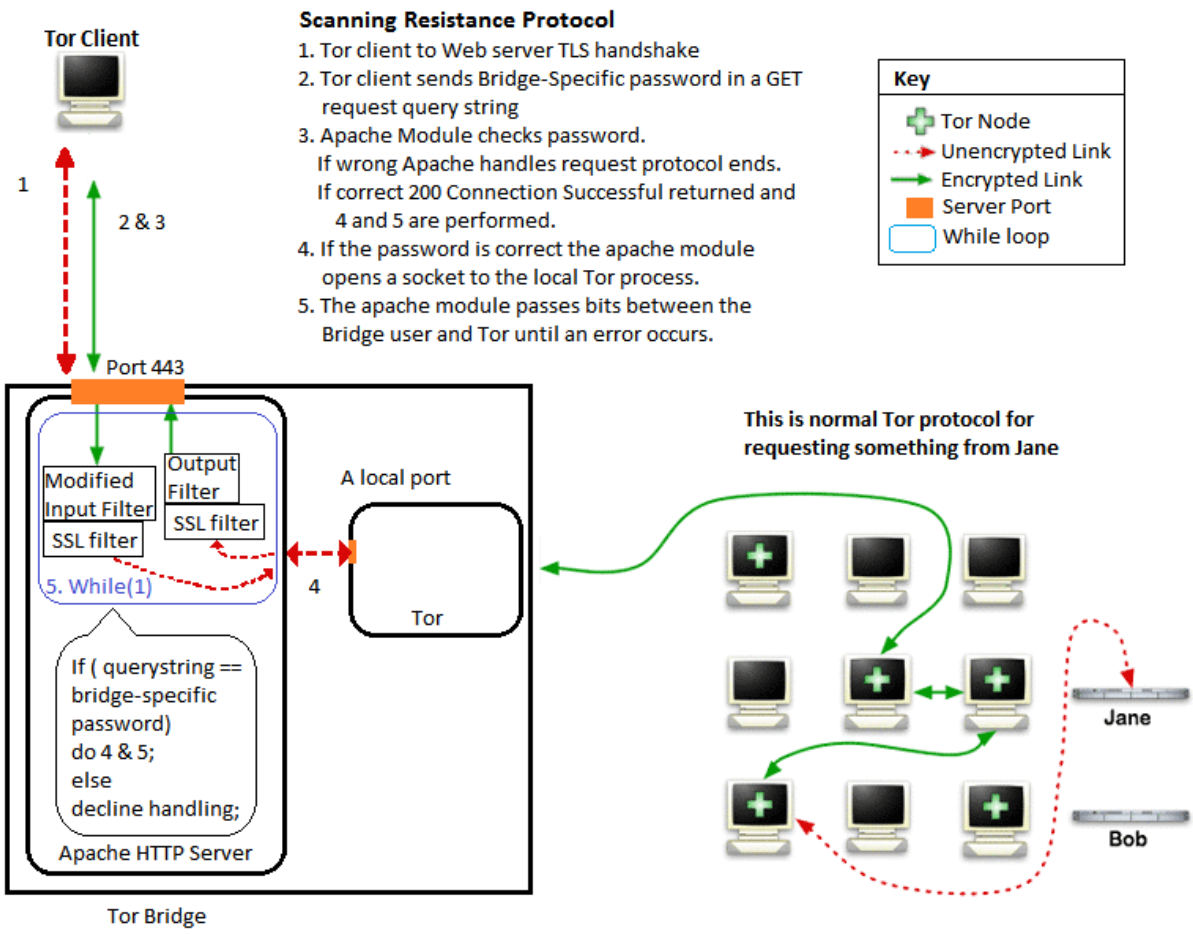

Figure 2: Protocol Design

This system aims to prevent the detection of Tor bridges via port-scanning. Another problem the system attempts to solve is concealing Tor's SSL handshake by performing a web-browser-looking SSL connection with the web server. Lastly, we want to make the system usable enough that it will be easy to widely deploy.

The protocol starts with the bridge operator generating a bridge-specific password. This bridge-specific password is given to trusted bridge users alongside the current bridge connection information including the bridge's IP address and fingerprint. The password will be generated automatically during Tor start-up, around the same time when Tor generates

a bridge fingerprint.

Next the bridge operator runs a slightly modified Tor bridge on a local port. The Tor bridge is modified to talk with no SSL for efficiency, as SSL under SSL is redundant.

The bridge operator also runs Apache on the standard http and https port. (This protocol does not use http, but allowing normal http access keeps the server looking like an average web server would.) The Apache web server will have a module that accepts the bridge-specific password in a GET query-string under HTTPS.

The Apache module consists of two parts. The first is an input filter which reads data directly from the client socket without modifying it. This input filter replaces the normal input filter after password verification in the second part.

The second part is the request processing handler. This handler performs metadata processing (Figure 1) to verify that the request is an SSL wrapped GET request with a query string matching the bridge-specific password. If the request does not match these things, it declines to handle content generation and another content generator will handle it. If the request verifies correctly, a socket is created to the local Tor bridge process and the modules sets up the bit passing.

To set up bit passing we remove all filters from the input chain except for SSL, and insert our input filter below SSL on the chain. This allows us to call the filter chain whenever data shows up on the client socket, and receive decrypted data. Next we remove all filters on the output chain before SSL. This removes the HTTP handling filters and leaves us with a SSL encryption filter, logging filter, and the filter that writes data back to the client.

After this setup, we create a poll on both sockets and listen for data on either socket. If data comes in from the client, we pull it through the input filter chain and send it as-is to the Tor process socket. If data comes in from the the Tor process socket, we take the bits and push them as-is into the output filter chain.

The bridge user will run a modified Tor client. The Tor client will remove the normal SSL connection handshake and replace it with the following. First the Tor client will connect to the bridge operator's IP on port 443 (Apache's HTTPS port) and follow a web browser-identical SSL handshake with the Apache server. Upon establishing a successful SSL connection to the server, the client sends a GET request with the password as a query string. If the password is correct, the server responds with a status 200 "connection successful" response. The Tor client then follows the Tor protocol normally, under the SSL

connection to the web server.

The Apache web server then does the following. Upon receiving a correct password, the module passes bits to and from the SSL connection and the local Tor process. If the password is not correct, the module declines to handle the request making the response equivalent to a normal Apache response with the same request. This maintains bug comparability with Apache, as there is no way to tell that the module is handling requests unless the password is known. Therefore, an adversary simply knocking at 443 can only tell that Apache is running, not Tor.

When Apache reaches the state of bit passing between the Tor bridge and Tor client, the Tor protocol follows the same protocol that current Tor runs.

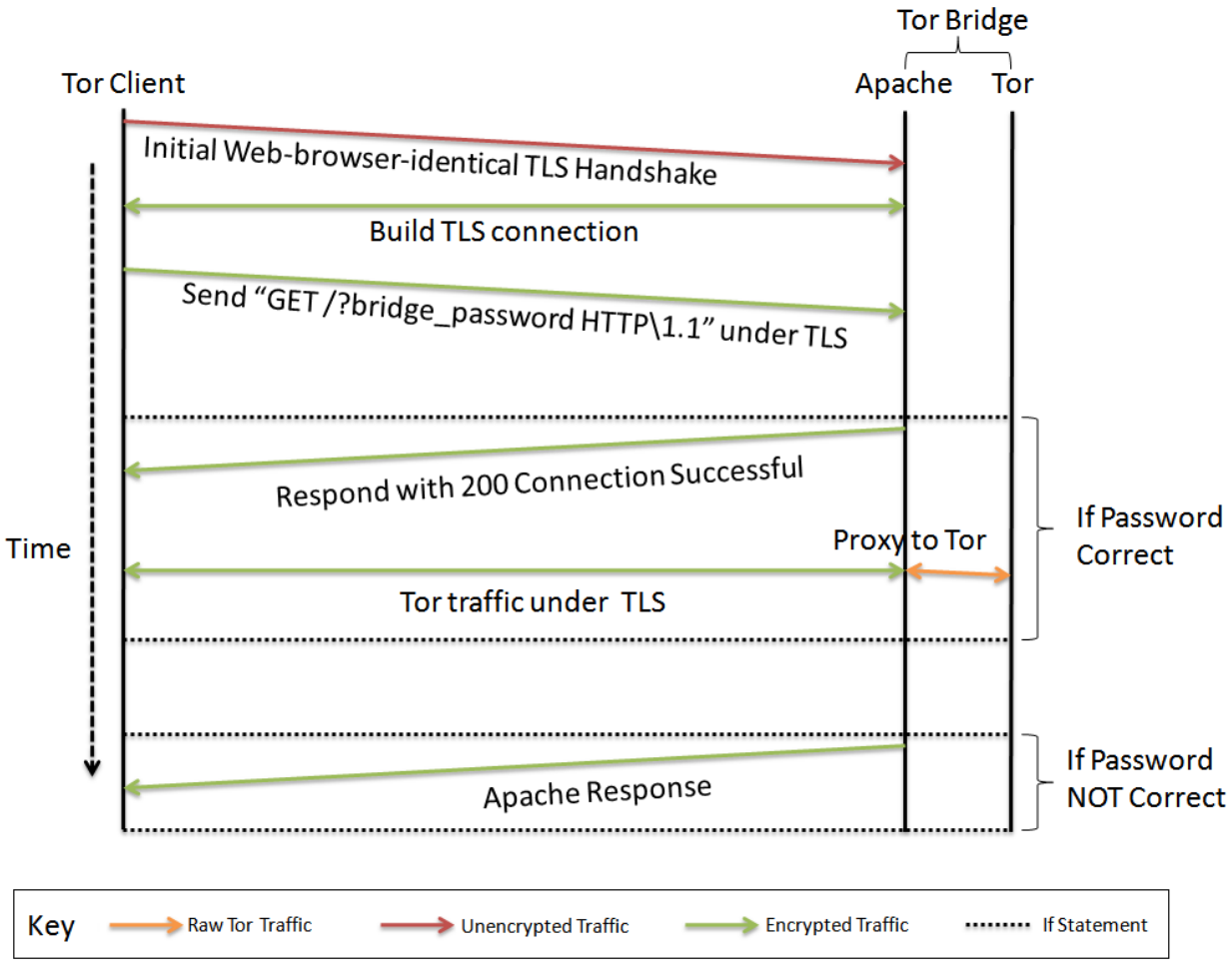

Figure 3: The port-scanning resistant bridge protocol in Arrow notation

Actual Implementation

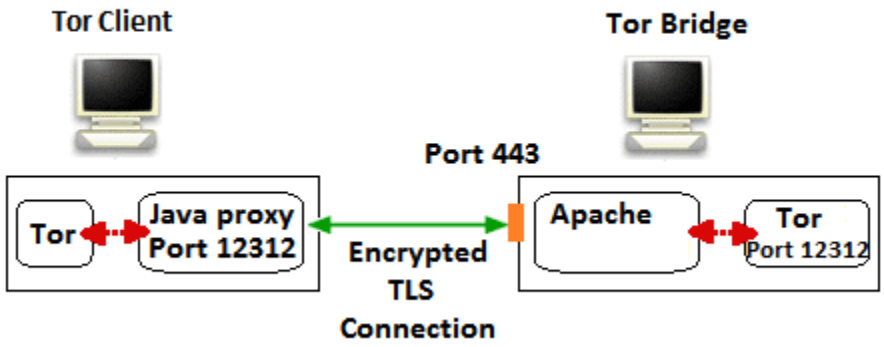

Figure 4: Actual Implementation

The Apache module was implemented as stated in the Protocol section, and works accordingly. But, due to time constraints and the large changes the protocol above would require in Tor, the Tor changes were replaced with a client side proxy written in java. This proxy accepts connections from the local Tor client. Once it receives a connection from the Tor client, it connects to the Apache web server on the host running the bridge. This connection starts with a TLS connection to port 443 and follows by sending the GET request with the bridge specific password. If the server replies with a 200 Connection Successful status, the java proxy switches to sending bits directly between the Tor client socket and Apache server socket. The Apache server socket maintains the TLS encryption and the java proxy encrypts and decrypts data appropriately.

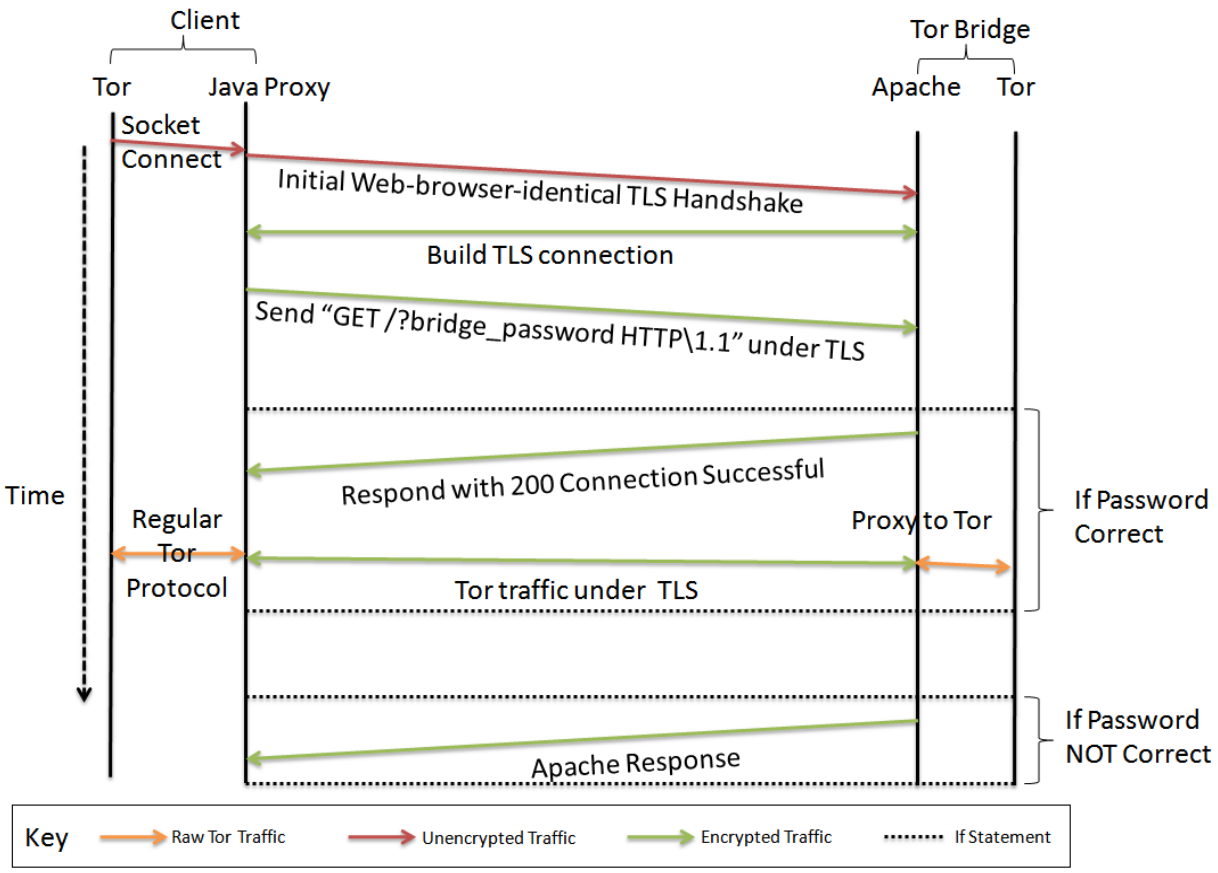

Figure 5: The implemented protocol with local java proxy in Arrow notation

Protocol Effectiveness

There are various attacks for detecting a Tor bridge or bridge user. Each attack has a different computational complexity associated with it. For example, port scanning bridges is relatively inexpensive compared to traffic analysis of encrypted SSL traffic based on packet size, quantity, and timing. Attack complexity is complicated further by false positives. Port scanning can have almost no false positives, because a port responding to Tor protocol must be running Tor. On the other hand, traffic analysis never knows with 100% certainty if the encrypted traffic is running Tor traffic underneath it. Table 1 shows the attacks that the protocol defends against and the bridge detection attacks

Attacks	Solved	Explanation
Port Scanning Bridge Identification	Yes	The protocol hides the Tor port locally behind Apache and a bridge-specific password.
Scanning for Tor SSL Handshake	Yes	Web browser-identical SSL handshake should prevent this attack.
Traffic Analysis of encrypted traffic based on packet size, quantity, and timing	No	Tor traffic is inherently different than web traffic with a server.

Table 1: Tor Attacks

Experimental Evaluation

It is important to note the extra overhead associated with the currently implemented protocol due to the extra layer of SSL and two proxies. As such, we ignore the rest of the Tor network to compare the difference between the normal bridge connection and the modified bridge connection. This is important because Tor circuits are highly variable in latency and bandwidth, and thus can create unreliable test results if one switches between circuits every test. (If our modified protocol gets implemented into Tor, the SSL layer between the Tor bridge and Tor client that normally exists would be removed. We do not need the extra layer of SSL, because we already have an SSL layer between the Tor client and the Apache web server.)

First, start-up times were monitored using the Tor testing library PuppeTor. Start-up times are how quickly it took a Tor node to create its first circuit. This requires the Tor bridge to send several hundred Tor relay addresses to the bridge user so the bridge user can decide who to make circuits with. A normal Tor process was set to only uses bridges and to connect to a bridge. This Tor process was timed while connecting to the Tor network until the first circuit was created. Then the modified process was tested the exact same way, connecting to the Tor network through a bridge 20 times. These were tested on the same virtual machine, connecting to the same bridge on a local area network. The results in Table 2 below show a little more than 3x differences between the start-up time of the

modified Tor protocol to the normal. This is most likely attributed to the extra java proxy and extra SSL neither of which will be in the final protocol, as well as the extra time required to pass bits from Apache to Tor and back. Another factor in this slowdown is the extra round trip times required to create the SSL and send the password for every TCP connection to Apache. It may be possible to modify the module so the TCP connections are restarted less often, but this is left to be addressed later based on how and why Tor makes new TCP connections.

	Average Start-up time	Standard Deviation
Normal Tor Protocol	6.453 sec	2.282 sec
Modified Protocol	20.313 sec	2.792 sec

Table 2: Normal vs Modified Start-up Times (20 trials each)

Next, Bandwidth was compared between the normal and modified Tor protocol using a fixed two hop circuit. The first hop was the normal or modified bridge on the same local area network as the client, the second hop was a relatively fast end-node("bach"). Next a bandwidth tester was used to download a 9.98KB file multiple times over the circuit. [5] The results show an almost a 3x decrease in bandwidth of the modified bridge compared to the normal. [Table 3] This slowdown is partially due to the extra layer of SSL and the extra proxying in the Apache module. However, the slowdown is much larger than it should be, which is likely due to the extra java proxy sitting between the client and the Apache server. When the java proxy functionality is incorporated into the Tor client, this slowdown should be reduced greatly. This 3x slowdown seems to match the slowdown in start-up times above.

	Average Bandwidth	Standard Deviation
Normal Tor Protocol	90.3kilobytes/sec	6.23kb/s
Modified Protocol	33.2 kilobytes/sec	.99kb/s

Table 3: Normal vs Modified Bandwidth Tests Through Tor Circuit: Picklejar(local bridge), bach(end-node) (28 trials each)

Finally, latency was tested using the program httping. [6] This program pings websites using a HEAD request and records the response time. Again a fixed circuit was used with the local bridge and a end-node. The latency recorded demonstrated a slight slowdown, which is expected due to the excess SSL and proxying. [Table 4] This does not seem unreasonable as the extra layer of SSL and proxying should cause a slight decrease in latency.

	Average Bandwidth	Standard Deviation
Normal Tor Protocol	993ms	289ms
Modified Tor Protocol	1195ms	191ms

Table 4: Normal vs Modified Latency Tests Through Tor Circuit: Picklejar(local bridge), bach(end-node) (100 trials each)

Problems and Future Directions

There are several problems this protocol does not solve. One problem is the protocol currently offers no defense against man-in-the-middle attacks when sending the plaintext password under TLS. A Tor client would need to verify that the SSL certificate is from the Tor bridge it thinks it is connecting to before sending the password. This becomes harder because we are using self-signed certificates, thus bridges cannot check if the certificate is from the bridge host or some man-in-the-middle. Another possibility would be for the server and bridge to hash the password with a nonce. It is hard to share a nonce without giving away that the server is running the Tor modules. Time is difficult to use due to large asynchrony between individual clocks on the internet. One possibility is to add some sort of request to the server before sending the password. However, maintaining bug-compatibility and making the web server look as average as possible hinder this option so this problem is left open.

Another issue with self-signed certificates is that even though there are plenty of misconfigured TLS certificates on the web, a country could filter access to any web server using self-signed certificates, thus blocking this protocol.

The protocol also fails if a firewall blocks HTTPS or connections to port 443, or can be severely limited if the firewall blocks all web servers on DSL and cable modems. These issues are both much harder to solve, but due to the harshness of these firewall rules on internet users, are less immediate threats.

Another important problem concerns what content the Apache server hosts on it. If people setting up Apache leave the main page to the default "It Works!" webpage, a filter could be implemented to block all websites lacking content. A user could host a variety of content to disguise the web site as a legitimate site. One problem is the site needs some reason to have SSL enabled, as many sites do not have SSL enabled. A secure login page or an online payment page are two of the biggest reasons a website would have SSL enabled. Therefore one could imagine setting up some sort of fake small business web site, or something similar to disguise the bridge. At the same time the websites need to be varied enough between bridges that someone could not scan for similar images or html to detect the bridges.

Conclusion

This protocol successfully prevents bridge port-scanning by hiding bridges behind an Apache web server and only allowing access to people with the bridge-specific password. This protocol also protects detection of Tor SSL handshakes by an active attacker by disguising it as a web-browser SSL handshake.

[1] Picturing Tor censorship in China - https://blog.torproject.org/blog/picturing-tor-censorship-in-china
[2] Tor: anonymity online - http://www.torproject.org/overview.html.en
[3] Scanning resistance: making bridges more subtle - https://git.torproject.org/checkout/tor/master/doc/design-paper/blocking.html#tth_sEc9.3
[4] Request Processing in Apache - http://www.apachetutor.org/dev/request
[5] Bandwidth Tester 0.5.9 - https://addons.mozilla.org/en-US/firefox/addon/178
[6] httping - http://www.vanheusden.com/httping/