CS262 Final Project: Tor-Based Privacy Preserving Network

Hileamlak Yitayew & Jeremy Zhang

May 7, 2023

GitHub Repository: https://github.com/hileamlakB/tor2

1 Introduction

1.1 Project Motivation

Tor stands for "the onion router," referring to the layers of encryption (like the layers of an onion) that a packet travelling the Tor network has such that no intermediate nodes can understand the underlying payload as well as the source and end points of the communication. Originally designed by Paul Syverson, Michael Reed, and David Goldschlag of the US Naval Research Laboratory, the goal of Tor was to allow for privacy and anonymity among participants within a distributed network.

Such a simple promise of privacy would have use cases ranging from mundane and trivial to highly sophisticated and very illegal. At its core, Tor's promise of anonymity can ensure an end user's safety. During times of civil unrest or international conflict, such as the Arab Spring in 2011 or more recently the war in Ukraine, civilians and soldiers alike have often relied on anonymity networks to safely communicate with each other or with their families outside of the country.

In popular culture, Tor has unfortunately become synonymous with the Silk Road and other nefarious activities conducted over the dark web. After all, criminals have a lot to gain from being anonymous online. Despite this, we believe that Tor is still worth studying due to its interesting distributed system setup. In addition, we hope that our project can help clarify misunderstandings surrounding Tor among students. This final report will serve as a blend of an engineering notebook and final paper, with the goal of documenting how we built a simplified version of Tor, some of the design trade-offs we made, and what we learned along the way.

1.2 Tor Background

To understand how the overall system works, we must first understand how onion encryption/routing works. At a high level, onion routing transmits an encrypted payload across the network, with each intermediate node decrypting (peeling away) one layer of the encryption, much like how layers of an onion have to be peeled away one at a time to reach the center of the onion.

For a Tor system with one client, one server, and three intermediate nodes (one guard, one relay, one exit), the process for communications is as follows. At the start of the process, the client (who wishes to send a message to the server) encrypts the message using keys k_1, k_2, and k_3. The client then sends this message to the guard node, which only knows key k_1. The guard node is able to decrypt the outer layer of the payload using k_1, which reveals that this "decoded" payload should be sent to the relay node. The relay node then decrypts the next layer using k_2 and sends the packet to the exit node. At the exit node, the packet is decoded using k_3, which reveals the actual payload. At this point, the exit node will interact with the desired server based on the contents of the payload. Upon receiving a response from the server, the response is then encrypted in the reverse order it was decrypted and sent back to the client. Note that k_1, k_2, and k_3 are agreed upon between the client and the relay node ahead of time, usually using some cryptographically secure key exchange method.

In addition to the onion routing scheme, Tor also provides a global network of relay nodes, a registry detailing the status of these relay nodes, and its own web browser that interacts securely with the Tor network.

1.3 Project Goals

Since building out an exact replica of the Tor network would require advanced knowledge of distributed systems, cryptography, and networking, we sought to build a simplified, proof-of-concept version of Tor to understand the process of sending an onion encrypted message across a distributed network. As such, here are the items we implemented in this project:

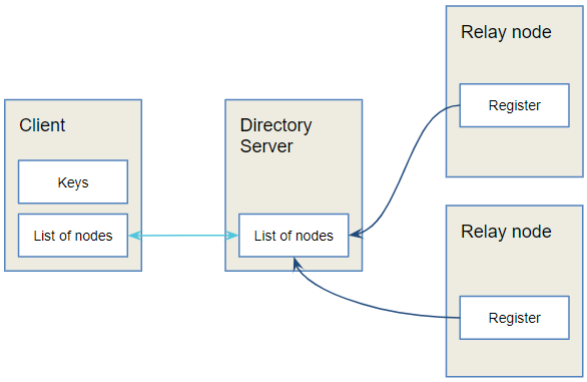
- Client, server, 3 intermediate/relay nodes (at least 1 entry node, 1 middle node, 1 exit node)
- Onion encryption – intermediate nodes do not see contents of message
- Basic fault tolerance - directory server detects relay node failures and updates the relay node directory accordingly
- Secure key exchange mechanism

The following items are out of scope for this project:

- Tor web browser
- Packet routing and congestion control

2 Design

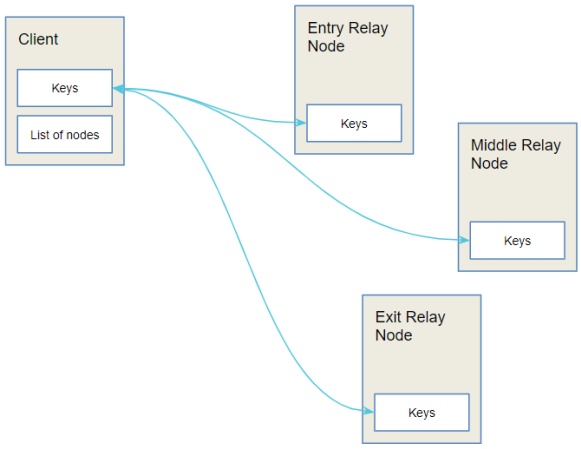
2.1 Directory Server Architecture



First, we implemented the directory server. The directory server has the main function of keeping track of the list of currently active relay nodes. When a relay node is created, it registers itself with the directory server by telling the server its address and type. When the client is started, the client will contact the directory server to obtain a list of all available relay nodes, their types (guard, middle, or exit), and their addresses.

For the initial demo in class, we decided to hard code three relay nodes (one of each type) and their addresses into the directory server list to save time and focus on the actual onion encryption and routing. However, the complete implementation, we are submitting, contains mechanisms for relay nodes to actually register themselves with the directory server.

2.2 Key Exchange Mechanism



The key exchange algorithm in our tor client is a crucial part of establishing secure communication between the client and the relay nodes in the network. Here's high-level overview of the key exchange algorithm, including a discussion on symmetric and asymmetric encryption, and the rationale for using RSA for encrypting symmetric keys:

1. **Symmetric and Asymmetric Encryption**: There are two types of encryption used in our tor implementation, symmetric and asymmetric encryption. Symmetric encryption uses the same key for both encryption and decryption, while asymmetric encryption uses different keys (public and private keys) for encryption and decryption. Symmetric encryption is generally faster and suitable for encrypting large amounts of data, while asymmetric encryption provides better security by allowing the separation of encryption and decryption keys.

2. **Key Generation**: The client generates a unique RSA key pair (public and private keys) for each relay node it wants to communicate with. This ensures that each layer of communication is encrypted with a different key, utilizing asymmetric encryption.

3. **Key Exchange**: The client sends its public keys to the respective relay nodes using the *ExchangeKeys* method. In return, the relay nodes send their public keys to the client. This exchange allows the client to encrypt messages using the relay nodes' public keys, and the relay nodes can decrypt these messages using their private keys. Similarly, the relay nodes can encrypt messages using the client's public keys, and the client can decrypt these messages using its private keys.

4. **Layered Encryption**: When the client wants to send a message through the network, it encrypts the message using a layered approach. First, the inner layer is encrypted using a symmetric encryption algorithm (AES) and an AES key, which is then encrypted using the exit node's public key (asymmetric encryption). Then, the middle layer is encrypted using another AES key and the middle node's public key. Finally, the outer layer is encrypted using a third AES key and the entry node's public key. This layered encryption ensures that each relay node only has access to the information needed to forward the message to the next node in the circuit.

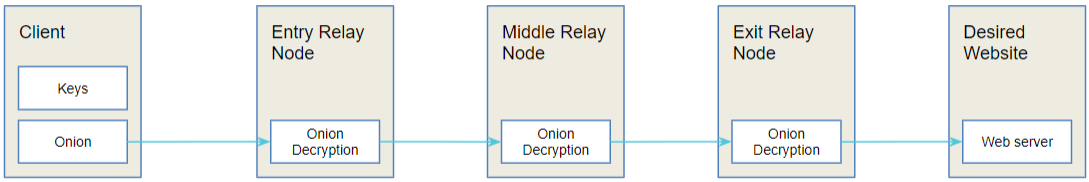
5. **Rationale for using RSA to encrypt symmetric keys**: The combination of symmetric and asymmetric encryption provides a balance between security and performance. Asymmetric encryption, such as RSA, is used to encrypt the symmetric AES keys, ensuring secure key exchange between the client and the relay nodes. This approach allows the client to securely share the symmetric keys with the relay nodes, while benefiting from the performance advantages of symmetric encryption for encrypting large messages.

6. **Layered Decryption**: When a message is received by the client, it decrypts the message in a reverse order, peeling off one encryption layer at a time. The client first decrypts the outer layer using its private key associated with the entry node, followed by the middle layer using its private key associated with the middle node, and finally the inner layer using its private key associated with the exit node. This process reveals the original message sent by the server.

This key exchange algorithm ensures end-to-end encryption between the client and the server while maintaining anonymity by routing the message through multiple relay nodes in the network. The combination of symmetric and asymmetric encryption techniques provides both security and performance advantages.

2.3 Onion Protocol

2.3.1 Sending a Message from the Client



The above diagram shows the process in which the client sends the encrypted onion (containing a request for a web server) across a series of nodes in the Tor network, eventually reaching the destination web server. Before sending the onion, the client must first construct it by encrypting the plaintext request (i.e. `send www.google.com GET`) using keys shared with the exit relay node, then encrypting that encrypted message along with the address of the exit relay node using keys shared with the middle relay node, and finally encrypting that encrypted message along with the address of the middle relay node using the keys shared with the entry relay node.

When the client sends the encrypted onion to the entry node, the entry node is only able to decrypt the outer layer of the onion, which reveals to it another encrypted message along with the address of the middle node (so the entry node knows where to forward the encrypted message). Upon receiving the partially decrypted (but still encrypted) onion, the middle node is able to decrypt the next layer of the onion, which reveals to it another encrypted message along with the address of the exit node. Finally, once this encrypted message reaches the exit node, the exit node decrypts this message, revealing the plaintext request. The exit node then contacts the appropriate web address or otherwise interacts with the internet as the client wishes.

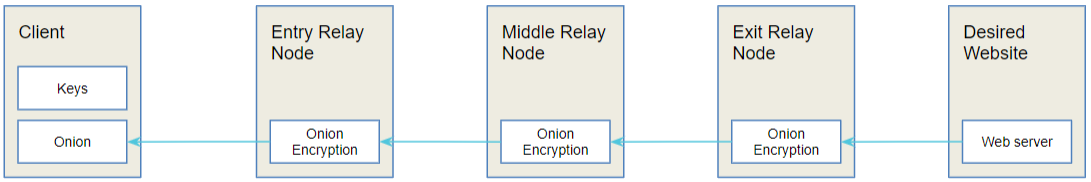
Although not perfect, this system is able to maintain the privacy of the client and their request. While the entry node knows that the request originated from the client, the entry node is unable to see the contents of the request. Although the exit node is able to see the plaintext request, it does not know the origin of such request. In our specific example, since there is only one client, it would not be difficult to figure out who the client was since the client exchanged keys with the exit node in the beginning. However, in a real world Tor network with potentially millions of users, the exit node would have a significantly harder time figuring out which client made the request, thereby protecting the privacy of the client.

Here is a pseudocode used to construct the onion and send it to the entry node. The code could be found in `client.py`,

Algorithm 1 SendMessage

procedure SENDMESSAGE($url, request\_type$)
$inner\_aes\_key \leftarrow random\_32bytes$
$inner\_msg \leftarrow create\_inner\_message(request\_type, url)$
$inner\_iv,\ inner\_onion \leftarrow aes\_encrypt(inner\_aes\_key, inner\_msg)$
$x \leftarrow rsa\_encrypted\_inner\_aes\_key$
$middle\_aes\_key \leftarrow random\_32bytes$
$middle\_msg \leftarrow create\_middle\_message(inner\_onion, exit\_relay\_address, x, inner\_iv)$
$middle\_iv,\ middle\_onion \leftarrow aes\_encrypt(middle\_aes\_key, middle\_msg)$
$outer\_aes\_key \leftarrow random\_32bytes$
$outer\_msg \leftarrow outer\_message(middle\_onion, ...)$
$outer\_iv,\ outer\_onion \leftarrow aes\_encrypt(outer\_aes\_key, outer\_msg)$
$outer\_aes\_encrypted\_key \leftarrow rsa\_encrypt(entry\_relay\_public\_key, outer\_aes\_key)$
$entry\_stub.ForwardMessage(ProcessMessageRequest(outer\_onion, ...))$
end procedure

2.3.2 Receiving a Message from the Server



The above figure shows the process of receiving information back from the desired web server. As the diagram suggests, the steps carried out in the previous section are basically performed again, except this time in reverse. First, the desired information from the web server is returned to the exit relay node. The exit relay node encrypts the web server response and sends it to the middle node. The middle node then adds its encryption and sends the encrypted payload to the entry node. The entry node then adds one final layer of encryption and returns this encrypted onion to the client.

Here is a pseudocode used to add a layer of encryption to the web server response as it works its way back through the Tor network and to the client. The code could be found in `relay_node.py`,

Algorithm 2 BackwardMessage

procedure BACKWARDMESSAGE($request, context$)
    $return\_address \leftarrow get\_return\_address(request.session\_id)$ # *from internal data structure*
    $aes\_key \leftarrow random\_32bytes$
    $msg \leftarrow create\_backwards\_message(request.encrypted\_message, request.encrypted\_key, request.iv)$
    $iv,\ encrypted\_message \leftarrow aes\_encrypt(aes\_key, msg)$
    $encrypted\_key \leftarrow rsa\_encrypt(get\_session\_key(request.session\_id), aes\_key)$
    $msg \leftarrow ProcessMessageRequest(encrypted\_message, encrypted\_key, iv, request.session\_id)$
    if $relay\_type == ENTRY\_NODE$ **then**
        $send\_to\_client(return\_address, msg)$
    else
        $send\_to\_next\_relay(return\_address, msg)$
    end if
    return *ProcessMessageResponse()*
end procedure

2.3.3 Peeling the Returned Onion

After the onion encrypting the web server's response is returned to the client, the client will deconstruct the onion by decrypting it one layer at a time, following a process that is largely the reverse of the process of constructing the onion.

Here is a pseudocode used to deconstruct the onion returned from the entry node. The code could be found in `client.py`,

Algorithm 3 PeelLayers

procedure PEELLAYERS($request$)
    $aes\_key \leftarrow rsa\_decrypt(private\_key\_0, request.encrypted\_key)$
    $first\_layer \leftarrow aes\_decrypt(aes\_key, request.iv, request.encrypted\_message)$
    $first\_layer\_msg \leftarrow parse\_message(first\_layer)$
    $encrypted\_msg,\ encrypted\_key,\ iv \leftarrow first\_layer\_msg$
    $aes\_key \leftarrow rsa\_decrypt(private\_key\_1, encrypted\_key)$
    $second\_layer \leftarrow aes\_decrypt(aes\_key, iv, encrypted\_msg)$
    $second\_layer\_msg \leftarrow parse\_message(second\_layer)$
    $encrypted\_msg,\ encrypted\_key,\ iv \leftarrow second\_layer\_msg$
    $aes\_key \leftarrow rsa\_decrypt(private\_key\_2, encrypted\_key)$
    $third\_layer \leftarrow aes\_decrypt(aes\_key, iv, encrypted\_msg)$
    $third\_layer\_msg \leftarrow parse\_message(third\_layer)$
    $headers,\ content \leftarrow third\_layer\_msg$
end procedure

3 Implementation Decisions and Considerations

We made the following decisions regarding implementation:

gRPC for communication: We used gRPC as the communication framework to facilitate connections between the clients, relay nodes, and the directory server. This is because gRPC offers efficient, language-agnostic communication, allowing us to easily extend our project and implement additional services if needed.

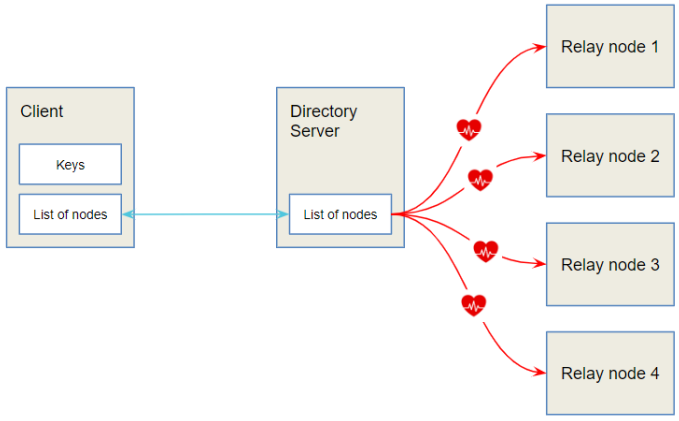
Protocol Buffers for service definition: we used Protocol Buffers to define the gRPC services and their methods. This provides a clear, structured format for defining services and messages, making it easy to understand and maintain. Both of us also had extensive experience working with gRPC and Protocol Buffers since we used them extensively on other design exercises in CS262. The full spec can be found in the `tor.proto` file in the repository.

Python libraries: Since the protocol buffers defined by gRPC did not automatically support the passing of complex, custom-defined Python dictionaries, we needed a way to serialize these data structures in a homogeneous fashion such that they can be reliably passed across the network. This was done fairly trivially using the Python Pickle library.

Single send message method for clients: The client sends messages through the Tor network using a single send message method. This simplifies the initial implementation while still demonstrating the core functionality of encrypted communication through the network.

4 Fault Tolerance

In this project, we have designed and implemented an $N + 1$ fault tolerance mechanism, for $N > 3$, where N represents the desired number of relays. The fault tolerance is achieved by enhancing both the client-side and directory server components to ensure robust communication even in the presence of relay node failures or unresponsiveness.



In the above figure, we see that the directory server plays a crucial role in maintaining the list of live relay nodes. It registers relay nodes upon their request and periodically pings them to verify their availability. If a relay node becomes unresponsive, the directory server removes it from the list, ensuring that only live nodes are provided to the clients. A relay node can register itself at a later time in case of a failure.

On the client-side, we have developed a strategy for selecting multiple circuits $(N + 1)$ to send messages through the network. Each circuit consists of at least three nodes: an entry node, a middle node, and an exit node. This design ensures that the message is routed through multiple paths, providing security

The client sends messages through the selected circuits and waits for a response. A response tracking mechanism has been implemented to monitor the successful reception of messages. If a response is not received within a specified timeout, the client assumes that one or more relay nodes in the circuit have failed or become unresponsive.

In such a case, the client requests additional relay nodes from the directory server and creates new circuits to retry the message transmission process. This mechanism guarantees that the client can still communicate through the network even if some relay nodes fail or become unresponsive.

By incorporating these enhancements in the client-side and directory server components, we have successfully improved the overall reliability and fault tolerance of the system. This design ensures that the network can maintain communication even in the face of relay node failures, providing a more resilient and robust solution for privacy-preserving communication.

5 Security Analysis

In this custom Tor implementation, several security measures have been taken to ensure the privacy and security of the communications. However, it is crucial to understand the possible attack vectors and how the system addresses them. Here is a detailed analysis of potential attack points and how our design mitigates these risks:

1. **Eavesdropping on client-relay communication:** An adversary may attempt to intercept the communication between the client and the relay nodes. To mitigate this risk, the implementation employs layered encryption (onion routing). The client encrypts the message multiple times, with each layer corresponding to a relay node in the circuit. Each relay node decrypts one layer, making it impossible for an eavesdropper to uncover the original message or the entire path.

2. **Compromised relay nodes:** An attacker may control one or more relay nodes in the network, potentially monitoring or tampering with the messages. The system's design ensures that even if an adversary controls some relay nodes, they cannot uncover the entire communication path or the original message. This is achieved by using a three-node circuit (entry, middle, and exit) and employing layered encryption. Controlling a

single node or even two nodes in the path is insufficient for the attacker to compromise the communication.

Note: If all three nodes in the circuit are compromised by an attacker, it is theoretically possible for the attacker to decrypt the message and trace the communication path. In such a scenario, the privacy and anonymity of the client would be at risk. It is important to acknowledge this limitation when discussing the security of the custom Tor implementation.

3. **Directory server compromise:** An attacker may target the directory server, which holds information about the available relay nodes. If the directory server is compromised, the attacker may provide the client with a malicious list of relay nodes, that the attacker have access to and we don't have any guards against this. One possible solution we were thinking about was to use multiple directory servers choosing relay nodes with a consensus mechanism like Paxos.

4. **Traffic correlation attacks:** An attacker may observe traffic patterns at multiple points in the network and correlate the timing and volume of traffic to deanonymize users. This attack is hard to guard against and is also a problem for the official Tor implementation. While our implementation does not specifically address this type of attack, it can be mitigated using techniques such as padding, constant-rate transmission, or introducing random delays to obscure traffic patterns.

In summary, our custom Tor implementation provides various security measures to protect the privacy and integrity of communications. While some potential attack vectors have been addressed, further research and development can help improve the system's robustness against advanced threats.

6 Conclusion and Future Work

In this report, we described our process in making a privacy-preserving distributed system inspired by the real-life Tor network. We detailed how we built our own architecture comprising of a directory server, client, and relay nodes, including how all of the parts of the system communicated with each other. In addition, we described our own advanced key encryption and exchange scheme, how we added fault tolerance to our distributed system, and where certain security vulnerabilities (present both in Tor overall and specific to our implementation) may be present and how they can be mitigated.

While this final project represents a complete implementation of a basic Tor-like privacy network/distributed system, we would have like to add more features if we had more time and resources. For example, we would have loved to conduct a more in depth examination into all of the failure modes of our distributed system, and then added more redundancy and recovery mechanisms to all of this distributed system's components to make it significantly more resistant to faults. We would have also loved to deploy our system onto a set of servers across the world to demonstrate it working truly in the real world, rather than just on a few of our computers on the Harvard network.

7 References & Acknowledgements

Here are some references we found to be extremely helpful in the preparation of our project:

The original Tor paper: Anonymous Connections and Onion Routing by Michael G. Reed, Paul F. Syverson, and David M. Goldschlag (1998).

The Tor spec: https://gitweb.torproject.org/torspec.git/tree/README.md

The Tor source code: https://git.torproject.org/tor.git

On directory authorities:
https://jordan-wright.com/blog/2015/05/14/how-tor-works-part-three-the-consensus/

On Tor security:
https://blog.torproject.org/research-problems-ten-ways-discover-tor-bridges/
Extensive Analysis and Large-Scale Empirical Evaluation of Tor Bridge Discovery by Zhen Ling, et. al. (2012).
ZMap: Fast Internet-Wide Scanning and its Security Applications by Zakir Durumeric, Eric Wustrow, and J. Alex Halderman (2013).

In addition, we would also like to thank Professor Waldo and the teaching staff of CS262 for such a wonderful semester. Your knowledge, expertise, generous support, and entertaining yet informative lectures certainly made this class a highlight of our semester.