McGill University

Onion Routing: A Quest for Internet Anonymity, and Perhaps also for Unparalleled Brilliance

ECSE 414 Final Report

Camilo Garcia La Rotta (260657037)
Dennis Liu (260581270)
Stuart Mashaal (260639962)
Spiros-Daniel Mavroidakos (260689391)
Jastaj Virdee (260689027)
Harley Wiltzer (260690006)
Edward Zhao (260690376)

December 6, 2017

Contents

1. Introduction and Background

Suppose it is desired to communicate sensitive information across the Internet to a trusted source. Of course, it should not be possible for an arbitrary observer to read this information. The problem of hiding information can be solved by encrypting data, taking advantage of the advances in the field of cryptography, so that only the trusted source may be able to unscramble (and thus effectively read) the message that was sent. But what if this communication was so sensitive that it is even desired that no arbitrary observer should even know that the source and trusted destination are communicating with each other? This presents another interesting problem that goes beyond cryptography – in fact, it may only be solved by the design of a clever communication protocol.

The desire to participate in anonymous communication over the Internet is quite popular nowadays, and many choose to accomplish this by communicating through a Virtual Private Network (VPN). This works by sending all messages through an intermediate server, so that observers can only see that the sender is sending messages to the VPN server, and therefore cannot tell where the sender's desired destination is. Moreover, the destination itself is receiving messages from the VPN server, so observers monitoring the destination cannot tell who the sender is. On the surface, it may seem like this method does, in fact, provide a means of anonymous communication. Since no observer can know the source and destination of a message sent through a VPN, does that not imply that anonymity was achieved? Unfortunately, although arbitrary observers cannot discern the source and destination of a message, there is one party that may retain this information, and that is the VPN server itself. Users of VPN services must trust that the owner of the VPN server has no malicious intentions, and does not, for example, keep logs of the packets it receives. Although many users may trust the VPN servers they use, it is important to note that the VPN service does not, in fact, provide complete anonymity.

This report will explore and provide details of the implementation of the Onion Routing protocol, a method of communication that was shown to provide complete anonymity [1]. Onion Routing may be seen as an extension of the VPN scheme described above, which makes creative use of several intermediate servers to hide the endpoints of a communication channel from any potential observer. An Onion Routing network consists of a set of *onion routers*, denoted by $\mathcal{O}$, and a *directory node*. The directory node is responsible for knowing the details of all onion

routers and selecting a random path of n onion routers, $\mathcal{P} \triangleq \{o_1, \ldots, o_n\} \subseteq \mathcal{O}$ for a user to communicate through. Given $\mathcal{P}$, the sender sends his message to o_1, who relays the message to o_2, and so on, until o_n receives the message and forwards it to the destination. Therefore, no intermediate node in $\mathcal{P}$ communicates with both the sender and the receiver of a given message, assuming $n > 1$ (note that, for $n = 1$, the scheme collapses to the VPN paradigm described above). However, so far, this scheme *does not* provide complete anonymity. Although no router in $\mathcal{P}$ directly communicates with the sender and destination, the routers in $\mathcal{P}$ must still know how to route a message across $\mathcal{P}$, meaning its messages must contain the addresses of all routers in $\mathcal{P}$. If a router knows $\mathcal{P}$, the owner of the router may trace the messages it forwards to the destination! Thankfully, Onion Routing solves this issue by layering encryption on messages in such a way that all routers in $\mathcal{P}$ can only know the address of the server before it and the server after it in the message's path. This prevents any router or observer from gaining full knowledge of $\mathcal{P}$, which effectively hides the endpoints of a communication channel.

It is up to the designer of the Onion Routing Protocol to come up with a way of designing the encryption scheme that hides $\mathcal{P}$ from observers. Ultimately, this scheme will be the backbone of the security of the protocol. In the implementation described in this report, this was accomplished by an elegant virtual circuit establishment procedure, which was inspired by a establishment process suggested by Reed, Syverson, and Goldschlag [2] but heavily modified, as will be described later on in the report.

All in all, the Onion Routing scheme ideally provides some drastic improvements to the VPN scheme with respect to security:

1. The virtual circuit establishment phase eliminates the possibility of any router or observer directly knowing both the sender and destination of a message
2. A large amount of available onion routers on the network provides severe difficulty of guessing a message path
3. If there is lots of traffic in the network, it is infeasible for an observer to try to trace the path of a message

Therefore, Onion Routing is a very promising technique for providing complete anonymity of a communication channel. However, it is also a drastically more complex protocol then communicating through a VPN, for example. The added complexity may cause many considerable design challenges, performance reductions, and practical limitations, which may ultimately affect its feasibility.

The remainder of this report will describe the implementation details of such a protocol, and will discuss the design challenges that were faced. Furthermore, a thorough *exposé* of the limitations of the protocol, as well as the costs of mitigating these limitations, will be given. Moreover, performance details such as the added delay imposed by multiple intermediate routers and encryption or decryption will be observed. Although the Onion Routing

protocol is promising, it remains to be seen if it is powerful enough for practical use. This report aims to determine if the proposed Onion Routing scheme can be used in practice, and will provide discussion concerning the cost of improving its practicality and the practicality of anonymous communication altogether.

2. Methodology

Onion Routing is a general enough concept that particular implementations of it must, on their own, make and incorporate many design choices about communication protocol and software architecture. For this reason, this section of the report serves to explore the technical details of this implementation's protocol and architecture, as well as explain, where applicable, the design choices behind these details.

2.1 Protocol Design

Say some sender $\mathcal{S}$ wishes to send a message to some recipient $\mathcal{R}$ through our Onion Routing Network implementation $\mathcal{O}$. First, $\mathcal{S}$ must establish a virtual circuit through $\mathcal{O}$ over which to communicate with $\mathcal{R}$. Reed, Syverson, and Goldschlag suggested a separate process for establishing the connections between all routers involved in a communication channel [2, 3], which was a very practical implementation idea that drastically simplified the routing of data messages and elegantly streamlined the circuit establishment process we ended up with which is as follows:

1. Acquire a path $\mathcal{P}$ from the *directory node* $\mathcal{D}$
2. Negotiate symmetric encryption keys with each node $o_i \in \mathcal{P}$

Once $\mathcal{S}$ has established $\mathcal{P}$ in $\mathcal{O}$, it can then wrap a message M intended for $\mathcal{R}$ in an *onion* before sending the onion through $\mathcal{P}$ to $\mathcal{R}$. The exact details of the onion data structure will be illucidated shortly, but for now it shall suffice to say that an onion is a message that is incrementally encrypted by $\mathcal{S}$ such that each node o_i can decrypt the message before passing it along to o_{i+1}.

To begin the circuit establishment process, $\mathcal{S}$ must acquire $\mathcal{P}$ from $\mathcal{D}$. However, it must do so securely so that no observer can know $\mathcal{P}$. Fortunately, $\mathcal{D}$, just like every onion router in $\mathcal{O}$, has an assymetric keypair $(pub_{\mathcal{D}}, priv_{\mathcal{D}})$. So, $\mathcal{S}$ generates a random symmetric key $sym_{\mathcal{S},\mathcal{D}}$. $\mathcal{D}$ will later use this to encrypt the path $\mathcal{P}$ that it will send to $\mathcal{S}$. Next, $\mathcal{S}$ encrypts $sym_{\mathcal{S},\mathcal{D}}$ with $pub_{\mathcal{D}}$ and sends that to $\mathcal{D}$. $\mathcal{D}$ then decrypts with $priv_{\mathcal{D}}$ so that now both $\mathcal{S}$ and $\mathcal{D}$ share $sym_{\mathcal{S},\mathcal{D}}$. Now that it shares a symmetric key with $\mathcal{S}$, $\mathcal{D}$ constructs a path $\mathcal{P}$, uses $sym_{\mathcal{S},\mathcal{D}}$ to encrypt a message containing $\mathcal{P}$ and sends the encrypted message to $\mathcal{S}$. To complete its path acquisition, $\mathcal{S}$ decrypts $\mathcal{D}$'s message to get $\mathcal{P}$.

Having finished acquiring $\mathcal{P}$ from $\mathcal{D}$, $\mathcal{S}$ can move on to step 2 of the circuit-establishment process: negotiating symmetric keys with each node $o_i \in \mathcal{P}$. $\mathcal{S}$ negotiates symmetric keys with all $o_i \in \mathcal{P}$ just as it did with $\mathcal{D}$: by generating the symmetric key randomly and then encrypting it with its recipient's public key before sending it. To clarify, this means that for each onion router o_i in $\mathcal{P}$:

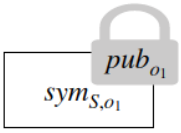
1. $\mathcal{S}$ randomly generates the symmetric key $sym_{\mathcal{S},o_i}$
2. $\mathcal{S}$ encrypts $sym_{\mathcal{S},o_i}$ with pub_{o_i} and sends that to o_i
3. o_i decrypts with $priv_{o_i}$ to finally share $sym_{\mathcal{S},o_i}$ with $\mathcal{S}$

Notably, unlike key negotiation with $\mathcal{D}$, it is unaceptable for $\mathcal{S}$ to do key negotiation with each $o_i \in \mathcal{P}$ directly. If $\mathcal{S}$ negotiates keys with each $o_i \in \mathcal{P}$ directly, malicious observers can know the identities of all nodes $o_i \in \mathcal{P}$. It was acceptable for observers to know that $\mathcal{S}$ was communicating with $\mathcal{D}$ because an Onion Routing Network does not (and maybe even cannot) attempt to hide the fact that $\mathcal{S}$ is trying to communicate through it, which is all that is revealed by observing $\mathcal{S}$'s communication with $\mathcal{D}$. Conversely, the anonymity of the nodes $o_i \in \mathcal{P}$ is crucial to the correctness of the Onion Routing implementation; if all $o_i \in \mathcal{P}$ can be known to any observer of $\mathcal{S}$ or the network, then any observer can watch messages leaving the last node in $\mathcal{P}$ to determine the identity of $\mathcal{R}$.

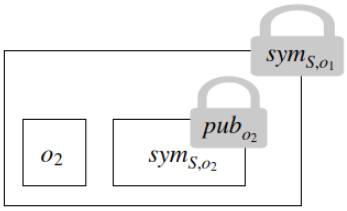
Therefore, $\mathcal{S}$ must negotiate symmetric keys with each $o_i \in \mathcal{P}$ *indirectly*, such that observing $\mathcal{S}$ during its circuit establishment cannot reveal every $o_i \in \mathcal{P}$. Using the example path $\mathcal{P} = \{o_1, o_2, o_3\}$, $\mathcal{S}$ would first negotiate a key with o_1, then would do the same with o_2 but *through* o_1, and finally would negotiate a key with o_3, but through both o_1 and o_2. But how can $\mathcal{S}$ negotiate a key with some node *through* other nodes? Using an *onion* data structure. Given the example path $\mathcal{P} = \{o_1, o_2, o_3\}$, let's walk through how $\mathcal{S}$ negotiates with all 3 nodes in the path *incrementally* using onions so we can have a complete view of the circuit establishment procedure.

First, $\mathcal{S}$ will negotiate a key with o_1 just as it would with $\mathcal{D}$, since o_1 is the first node in $\mathcal{P}$. Then $\mathcal{S}$ must negotiate with o_2 through o_1. As usual, $\mathcal{S}$ will start by generating a key $sym_{\mathcal{S},o_2}$ and encrypting said key using pub_{o_2}. For o_1 to relay this encrypted key to o_2, $\mathcal{S}$ must send o_1 something more – it must also send o_1 the identity of o_2 so that o_1 can know where to relay $\mathcal{S}$'s messages. Yet $\mathcal{S}$ cannot merely prepend the identity of o_2 to the encrypted $sym_{\mathcal{S},o_2}$ because anyone observing $\mathcal{S}$'s messages to o_1 would then know the identity of o_2, the node after o_1 in $\mathcal{P}$. However, if, after prepending the identity of o_2, $\mathcal{S}$ then encrypts with $sym_{\mathcal{S},o_1}$, $\mathcal{S}$ can securely have o_1 pass along the encrypted key intended for o_2. o_1 need only decrypt with $sym_{\mathcal{S},o_1}$ to see the identity of o_2 prepended to some encrypted content and forward that content to o_2. This data structure that $\mathcal{S}$ sends to o_1 is an *onion* (and a single-layer onion, at that). The onion is what allows o_1 to securely relay content to o_2. Each layer of the onion is defined by prepending the identity of o_{i+1} and encrypting with $sym_{\mathcal{S},o_i}$.

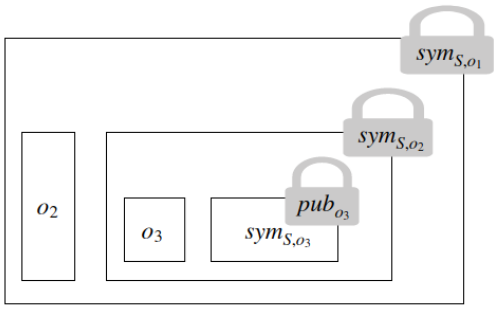
At this point is should be obvious how $\mathcal{S}$ will get the encrypted key $sym_{\mathcal{S},o_3}$ into the hands of o_3. $\mathcal{S}$ will prepend o_3 and encrypt with $sym_{\mathcal{S},o_2}$, and then prepend o_2 and encrypt with $sym_{\mathcal{S},o_1}$. Shown below is a graphical representation of the three messages sent by $\mathcal{S}$ to o_1 to establish the virtual circuit through $\mathcal{P} = \{o_1, o_2, o_3\}$.



Negotiate symmetric key with o_1 directly

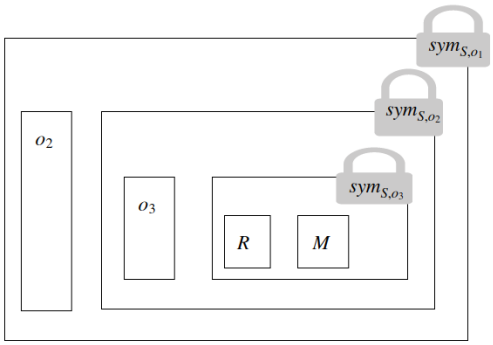


Negotiate symmetric key with o_2 through o_1 using single-layer onion



Negotiate symmetric key with o_3 through o_1 and o_2 using double-layer onion

Once the virtual circuit over $\mathcal{P}$ is established, $\mathcal{S}$ can send messages M to $\mathcal{R}$ freely by encapsulating each message in an onion. An example onion is shown below, using, once again, path $\mathcal{P} = \{o_1, o_2, o_3\}$.



Example of an onion sent from $\mathcal{S}$ through $\mathcal{P}$ to $\mathcal{R}$. In other words, $\mathcal{S}$ simply sends this onion to node o_1.

Having finished discussing the mechanics of virtual circuit establishment and messaging from $\mathcal{S}$ to $\mathcal{R}$ using onions, the only unexplored portion of the Onion Routing Communication Protocol concerns the mechanics of relaying messages from $\mathcal{R}$ *back* to $\mathcal{S}$. Thankfully, 'backwards-relaying' is exceedingly simple. Continuing with the example path $\mathcal{P} = \{o_1, o_2, o_3\}$, when o_3 receives a message M from $\mathcal{R}$, o_3 encrypts with $sym_{\mathcal{S},o_3}$ and relays to o_2 which then encrypts with $sym_{\mathcal{S},o_2}$ and relays to o_1 which then encrypts with $sym_{\mathcal{S},o_1}$ and relays to $\mathcal{S}$. Finally, $\mathcal{S}$ decrypts with $sym_{\mathcal{S},o_1}$ then $sym_{\mathcal{S},o_2}$ then $sym_{\mathcal{S},o_3}$ to reveal the message M from $\mathcal{R}$.

Though this protocol seems to correctly provide a means for $\mathcal{S}$ and $\mathcal{R}$ to communicate annonymously through $\mathcal{O}$, the protocol actually provides no anonymity at all under certain conditions. Specifically, if a third party could observe the entire network simultaneously and there were very few users (for example, only a single sender $\mathcal{S}$), Onion Routing would be of little use. Under those conditions, the third party would observe, for each message

M sent by $\mathcal{S}$ along $\mathcal{P}$ intended for $\mathcal{R}$, $\mathcal{S}$ send a message to o_1, then o_1 send to o_2, then o_2 send to o_3, and at last o_3 send to $\mathcal{R}$. Clearly, the usefullness of Onion Routing depends on the inability of third party observers to determine the last node in $\mathcal{P}$ for any message M from $\mathcal{S}$. Therefore, Onion Routing is only an effective means of anonymization when participants are numerous. When there are large quantities of simultaneous users in an Onion Network, an 'omniscient' observer would still be able to see all traffic flowing through the network but it would be infeasible to correlate the activity of the various nodes to the various users/senders $\mathcal{S}$. Indeed, a user of an Onion Routing Network is similar to a participant in a riot: anonymity is directly dependent on the number of accomplices.

Yet another condition which defeats the purpose of an Onion Routing network is when the network itself consists of too few routers. In this case, the observer need observe so few nodes that the set of possible recipients $\mathcal{R}$, which is always equal to the set of endpoints of outgoing connections from the network, is too small to provide practical anonymity.

Even when all the appropriate conditions are met, it is unclear whether a truly capable attacker can use statistical techniques to correlate outgoing nodes to a particular sender. Nonetheless, having investigated our implementation of an Onion Routing Protocol in fair detail, Onion Routing stands as a viable technique for providing anonymity to communications on the open internet.

2.2 Software Architecture and Implementation: TODO Camillo?

We leveraged Pythons socket and Threading library to avoid having to implement the lower level requirements of this project. The nodes that comprise the Onion network are: the Directory, the Router, the Client and the Destination. The directory and the router nodes sit in the inside of the network, only listening on predefined ports. Whereas the client and the destination sit on the edge of the network, being able to communicate on other ports. All the nodes in the network are multithreaded to support multiple parallel communication between different clients, destination pairs. The Directory node keeps track of participant router nodes and generates a random path of 3 nodes whenever a client node wishes to communicate through the network. The router node has 2 primordial function: to pass along the path a setup onion and to handle the bidirectional transfer of onions to the next and prior node in the path. The destination, nicknamed dummy destination, has the sole purpose of responding to any incoming message on an arbitrarily selected port. The client acts as the command line interface for the client using the network. Given a destination it will request a path from the Directory, setup the virtual circuit and build the onion to send to the destination. All the nodes on the network use a custom-made library for encryption, wich we named Stealth and provides convenient object for symmetric and asymmetric encryption. This implementation of the onion router is built upon TCP communication and relies on a globally defined socket for all internal nodes.

Our onion router network has the following implemented:

1. Directory node: Responsible for the specifying a path/ series of router nodes to pass through when a message is sent and to also send the respective keys for the routers

2. Onion router node: Responsible for peeling and encrypting messages depending if the message is in the forward or backward channel. Depending on how the routers are placed in a path, when a message is sent from one router to another, the receiver will send an acknowledgment message to confirm it has received the correct message.

3. Client node:

   (a) This is the node that will send the layered encrypted message in the forward channel

   (b) The client node will also send a message to all the nodes given by the directory node to set up a channel for it to send a message to the destination node

   (c) When it receives the encrypted confirmation message from the destination node, it will then decrypt the message to allow to user to see

4. Destination node: This is the node that will receive the complete message also send back a confirmation message in the backward channel. This message will have a layer of encryption added on at every onion router node until it has reached the client node to be decrypted

3. Results

Once the onion routing implementation was in working order, the next order of business was to ensure that it was functioning properly under various stresses, to determine the limitations of the system, and to ultimately analyze its practicality. Various stresses were taken into account, such as the performance effect of having several virtual circuits set up simultaneously and the behaviour of the communication system when faced with messages of various lengths. Furthermore, timing measurements were made to compare the setup time and round trip time of the proposed onion network compared to simple, garden variety message paths. The remainder of this chapter discusses the results that were found.

3.1 Behaviour with Several Simultaneous Virtual Circuits

It was shown previously that onion routing is only effective when there is lots of traffic within the onion network – this makes it infeasible for an eavesdropper to discern a particular user's message path $\mathcal{P}$. Therefore, it was imperative to ensure that the network performed appropriately when there are multiple simultaneous virtual circuits set up, which simulates multiple users sending data and causing traffic in the network of onion routers.

To attain a comprehensive examination of the system under these conditions, a first test involved the following steps:

1. Deploy three onion routers, so as to force all virtual circuits to pass through each onion router (thereby forcing each onion router to experience the effects of traffic)
2. Deploy a directory node
3. Launch several (> 10) clients set up to communicate through the onion network set up in steps 1 and 2
4. Send many messages from each of the clients through the network, and observe any behavioural oddities

The first run-through of this test failed miserably, due to a logical error in development that corrupted the symmetric keys on the onion routers. However, once this was fixed, the system proceeded to behave as normal, without any noticeable behavioural issues. Ultimately, this result was achieved fairly easily due to the well-planned *safe*

architecture that was implemented for multithreading in the onion routers. However, the *safe* architecture design was only conceived upon realizing the glaring *lacunae* in the implementation, brought to light by the development flaw mentioned prior.

This realization led the team to devise a new model for mutually-exclusive concurrent programming. Shared variables were either avoided or modified to create the following model for mutual exclusion, described by the lambda calculus:

$$\vdash \mathsf{mutex} : (\mathsf{unit} \xrightarrow{\aleph} \alpha) \xrightarrow{\omega} \alpha \tag{3.1}$$

Where $\aleph$ and ω are multiplicities, and $|\alpha| \triangleq \omega$. As shown by Niehren, Schwinghammer, and Smolka, this means that if the mutex (mutual exclusion) itself has multiplicity ω, it may be applied or copied arbitrarily many times [4]. This was a promising step towards making safer concurrent processes, and thus improved behavior with multiple simultaneous virtual circuits.

Further inspired by Niehren, Schwinghammer, and Smolka's work, a slight change to the archiecture of listening ports and key exchange ports was made to satisfy the following model of new port allocation:

$$\vdash \mathsf{newPort} : \mathsf{unit} \xrightarrow{\omega} (\alpha\mathsf{list}^{\omega} \times^{\aleph} \alpha \xrightarrow{\omega} \mathsf{unit})[4] \tag{3.2}$$

Where $\times^{\aleph}$ may be interpreted as a cartesian product of multiplicity $\aleph$. Niehren, Schwinghammer, and Smolka allude to a newChannel abstraction, which is remarkably similar to the virtual circuits that were already designed. Given these abstractions, it has been shown that proper implementation can lead to provable thread-safety [4].

Niehren, Schwinghammer, and Smolka recommend implementing abstractions for their researched $\lambda(\mathsf{fut})$ – a concurrent lambda calculus with futures. Given its promise and security implications, it was a good fit for the implementation of multithreaded onion routers. Using a conceptual lambda calulus as a foundation may seem eccentric – though in truth, all people that continuously *hanker* about a golden yesteryear implement bleeding-edge lambda calculi.

Based on what has been proven true about $\lambda(\mathsf{fut})$, and reinforced by the test results, it can be concluded that the onion routing system presented in this text is invulnerable to behavioural deficiencies due to increased virtual circuit density. This is a promising result in the validation of the onion network's power to preserve anonymity.

3.2 Message Size

It was very important to observe the effects that message size had on the onion network, as size limitations will play an important role in measuring the practicality of the system. Furthermore, it was important to note if any observation of message size could compromise the security of the system.

Firstly, messages were sent with increasing length until an odd behaviour was seen. The messages were taken from the language $\mathcal{L} = \{(\mathsf{abcdefghijklmnopqrstuvwxyz})^n | n \in \mathbb{N}$, onion network doesn't crash$\}$. When the network crashed, $n-1$ was noted and the message size was thereby incremented starting from $26(n-1)$ until a precise limit was found. It was seen that the maximum message size was 833 characters. Given that the developers hardcoded limits of 1024 bytes for message passing and the fact that there is overhead associated with the routing protocol, this result makes sense. More importantly, it is noted that messages sized at approximately 80% of the hardcoded limit can be accepted, larger messages can be easily accomodated by increasing this hardcoded limit.

It was also noted that message sizes generally increased as they travelled across the onion network. This is a potentially dangerous result, as with little traffic on the network message paths may be deduced. Through further testing, it has been seen that the base 64 encoding of encrypted data causes the increase in message length and not the encryption itself. Given more time, the developers could have used different encoding schemes (discussed in the Discussion section) to avoid this issue, and maintain safer message size properties.

3.3 Round Trip Time of Messages

Although the use of onion routing increases security and anonymity, there is a concern regarding the time needed for a message to be sent and received. The reason for this is instead of the message going from the client to destination and back, it must now also pass through several other nodes, and undergo encryption or decryption at each hop.

To go about determining whether the onion routing system took a long time to send and receive messages, we compared the timing taken by the onion routing system and checked whether it was noticeably longer than the timing of a normal TCP connection.

To compute the average time taken by the onion routing system to send and receive a message, the code of the system was edited so that it would print the times corresponding to when the message was sent and received. The sent time was then subtracted from the received time to determine how long it took for the message to reach the destination and return to the client. Ten messages were sent, and the average was taken of these times to determine the av-

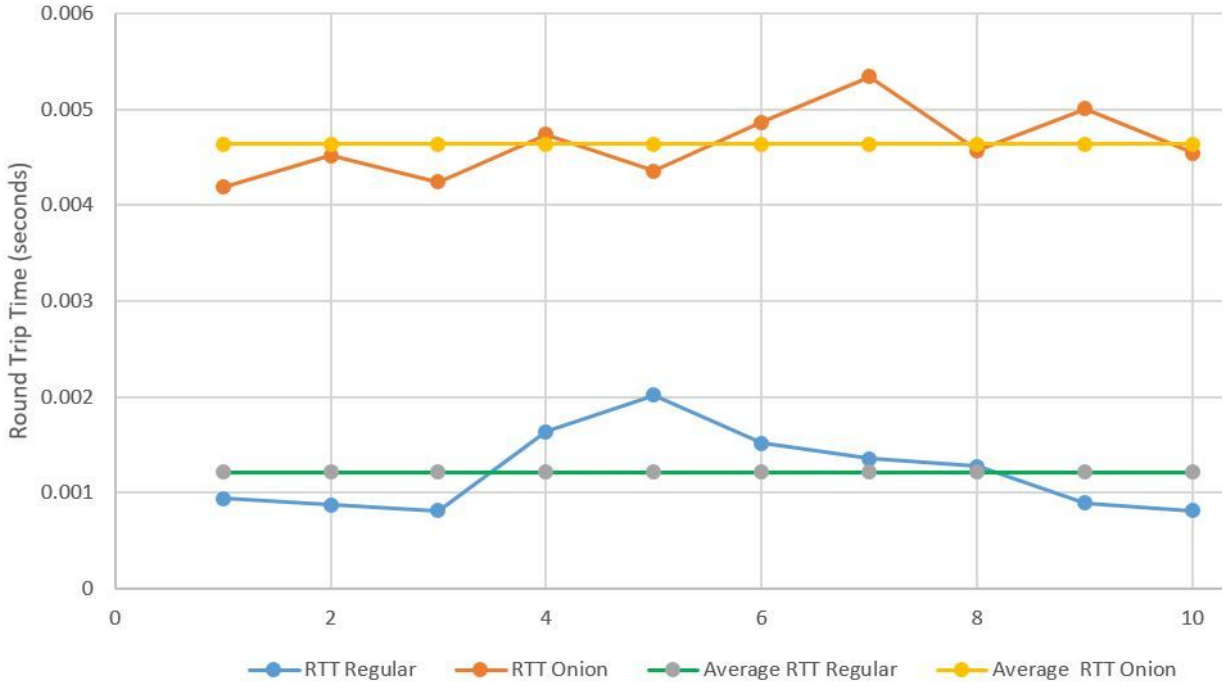
erage time taken by the onion routing system. The messages were of various sizes to ensure a more complete testing.

To compute the average time taken by a normal TCP connection, a simple python script was created to send messages from a client to a destination. Once again ten messages were sent, and the average was computed of these ten times. The ten messages that were sent were the exact same as the 10 messages that were sent on the onion network. This was to ensure that the only variable determining the average time was the type of connection (onion vs. TCP).

The following graph and table demonstrate the data collected:

Message	RTT TCP	RTT Onion
hi	0.000942	0.004191
hello	0.000875	0.004517
hello world	0.000814	0.004243
hello newman. Jerry	0.001639	0.004739
hello how are you	0.00202	0.004359
bye	0.001519	0.004865
hello how are? Good thanks how are you?	0.001353	0.00534
hi	0.001274	0.004572
⟨alphabet⟩	0.00089	0.00501
⟨alphabet x3⟩	0.00081	0.004543
Average	0.0012136	0.0046379

Figure 3.1: Round trip time of data through onion network compared to direct, unencrypted path



It can be seen that the difference between the round trip time of the onion network and normal TCP communication is approximately 3 milliseconds, on average. Although this may seem like a miniscule difference, the size of the messages sent in the test were small. Furthermore, it is important to consider not just the absolute difference, but the relative difference. It was shown that even for relatively small messages, as seen in Figure 3.1, the round trip time in the onion network was appromixately 3.8216 times greater than that of the regular TCP communication. Taking the average round trip time in regular TCP communication as a reference, the relative round trip time of the onion network communication is approximately 5.8225 dB, which is substantial. This shows that there is, in fact, a considerable price to pay for achieving anonymous communication. Of course, the particular price to pay depends on the quality of the servers hosting onion routers, and will further depend on other factors including message sizes and network congestion. The practicality of this reduction in speed will depend on the use case for the onion network as well. For relatively small messages, the network was shown to be relatively fast still, with a round trip time of just under 5 milliseconds. Therefore, for relatively small message sizes, the performance cost of using the onion network will be negligible for most applications.

4. Discussion

Since the beginning of the project, it was decided to split the duties across two teams: one team for developing the software, and another team for validating the software. The goal of the validation team was to perform a rigorous series of creative black box tests to effectively try to break the system. Under this methodology, the two teams observe the software from two different perspectives, which allowed for finding issues with the software in an efficient manner.

Though, even under this development paradigm, problems still arose like weeds in a garden. The first major design issue revolved around keeping ciphertext sizes constant throughout the network to avoid the possibility of a length attack, an attack allowing an eavesdropper to find the path of onion routers that a client is communicating through. Originally, the design for the onion routing involved sending header data (such as next node address, next node port, message type) under each layer of encryption. When the onion routers would peel off a layer, they would not propagate the header data forward (if they did, the last router could determine the message path), so the messages decreased in size over the network. Given the early implementation of the onion routing system, implementing fixed-size messages was an enormous challenge, because intermediate nodes could not add padding to data in such a way that both the padding is encrypted and the next node can decrypt the message, because each onion router only knows its own symmetric key. The first idea to solve this issue involved adding a large, random amount of padding to the plaintext at the originator. This way, tracing messages sizes becomes much less predictable, and thus performing a length attack becomes much more difficult. However, this solution required an enormous amount of overhead, and was more of a "hack" than a decent solution. The next idea to solve this issue was to give each router access to the previous node in the path's symmetric key. This way, when before propagating a message forward, the router may add padding (so as to fix the message size) and encrypt the new message with its own symmetric key. Then, when the next node receives a message, it can decrypt this layer of encryption and remove the padding. However, this solution was quite complex, as it also involved providing each onion router (and the client) with extra random number generators to keep the symmetric encryption objects in sync. Furthermore, the job of each onion router became far more complex, which negatively impacted performance. Finally, allowing each router to see the previous node's symmetric key is sloppy in design and could lead to further security issues. Fortunately, the solution to this issue came forth after a realization about the virtual circuit establishment process

that was described earlier. Since the virtual circuit establishment process opens all requisite TCP connections for the virtual circuit to function, the header data mentioned previously was no longer needed in order to route the message effectively! Therefore, the onion-creation algorithm and onion router protocols were simplified, and problem of decreasing message sizes was no longer an issue.
Unfortunately, constant-size messages still has not been implemented perfectly. Due to the base 64 encoding of encrypted data being sent over the network, message sizes now may increase slightly as they are propagated forward. Given more time, the development could have researched other encoding schemes (or perhaps avoided encoding altogether) to resolve this issue. Nonetheless, the current implementation has the best message size handling of the solutions suggested above, and is implemented by simpler and easily-readable code, which is a vast improvement.

For the testing team, the big question was "to automate or not to automate". The testing team oversaw making sure that the software worked, meaning that the encryption was working and that the messages were being delivered and received from host to destination and vice-versa. Initially, all the testing was done manually. The testers would manually set up every forwarding node, the destination node, the client node and the directory node. This would become time-consuming since there would often be 10 or more nodes working at a time to test the system at scale. A discussion took place and the decision to automate the deployment of the network was made. Two scripts were made to work in tandem. The first one, called `networkInitializer`, was a bash script that would `ssh` the user into many machines. The second bash script, called `executeNetwork`, would start up a given number of forwarding nodes along with a client node, directory node and destination node. Furthermore, another bash script called `testCases` was written to send messages sequentially instead of manually typing them in. This eliminated the redundancy of running regression tests and removed the boilerplater setup commands, effectively saving a lot of time.

A lot of work was completed for this project, but more features could have been implemented given more time. The current network implementation has a big flaw concerning fault tolerance. If any of the onion routers were to go down or fail to communicate, then all of the threads involved would raise uncaught exceptions, causing the involved onion routers to crash. If time permitted, fault tolerance would have been implemented and the nodes would be able to shut down in a graceful manner. This would mean that the routers would not raise uncaught exceptions and would propagate news of the circuit's failure to all participating nodes before leaving the circuit themselves. Also, the failure of any router in a virtual circuit would cause the circuit in question to shut down, and the originator clients involved would be signaled to establish new virtual circuits.

Another enhancement that could have been beneficial would be to have log files be generated whenever an error would happen in the network to simplify the debugging process. Furthermore, it would have been better to write the timing results to a file during the reception of the message instead of printing it to standard output.

All things considered, the tools such as git and the weekly meetings helped to keep the team on track throughout the entire process and ensured that every member of the team communicated when an issue or bug was found. The in-team communication allowed for bugs and design decisions to be dealt with swiftly and efficiently. Overall, although the onion routing implementation has its flaws, quasi-practical encrypted and "chaotic" communication was observed. Therefore, the onion network was shown to hide communication endpoint data from simple enemies. As mentioned above, there remains some changes to be made to fully secure the system, but the given implementation outperforms a VPN service with regard to anonymity over the Internet.

Appendices

A. A Brief Conspectus of Cryptography

The Onion Routing protocol described in this report makes extensive use of cryptography for encrypting and decrypting sensitive data. Encryption is the process of obscuring data in such a way that only the intended recipient of the data may read the data, via decryption. In order for this to be accomplished, the sender and the receiver of the data use calculated *keys*, which are similar in purpose to passwords, to compute the scrambled data and to retrieve the unscrambled data. Two main paradigms for achieving encryption and decryption are used in this implementation, and will be described below.

A.1 Terminology

The list below describes some terms related to cryptography that may be used sporadically in the report.

- **Plaintext**: Regular text that has not been encrypted.
- **Ciphertext**: Scrambled, incomprehensible text that is the product of encryption. Must be *decrypted* to be understood.
- **Encryption**: The calculated process of scrambling data to create ciphertext in such a way that it can be unscrambled only by a desired party.
- **Decryption**: The process of unscrambling ciphertext into plaintext.
- **Key**: A byte string that ultimately cryptographically identifies a communicating party, which is used to encrypt and/or decrypt data.

A.2 Symmetric Cryptography

The simpler form of cryptography that will be used is called "Symmetric Cryptography", and is characterized by the notion of the sender and receiver sharing a secret key, called a symmetric key. The scheme is defined by the tuple $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$. The KGen parameter is pseudorandom generator that generates a key $k \in \mathcal{K}$ with uniform probability over $\mathcal{K}$, where $\mathcal{K}$ is the keyspace, or the set of all keys. $\mathsf{Enc}, \mathsf{Dec}$ are families of functions,

where $\mathsf{Enc}_k : \mathcal{M} \to \mathcal{C}$ and $\mathsf{Dec}_k : \mathcal{C} \to \mathcal{M}$ are encryption and decryption functions for some key $k \in \mathcal{K}$, $\mathcal{M}$ is the message space (set of all possible messages), and $\mathcal{C}$ is the ciphertext space (set of all possible ciphertexts). Symmetric schemes are called symmetric because encryption and decryption are both done using the same key, so for all messages $m \in \mathcal{M}$,

$$\mathsf{Dec}_k(\mathsf{Enc}_k(m)) = m \tag{A.1}$$

This property is crucial for providing bidirectional encrypted communication without revealing the message's route, so symmetric encryption and decryption is a primary function of the onion routers. Furthermore, symmetric key encryption and decryption are asymptotically much faster than their asymmetric counterparts [5] (described in the next section). The downside, of course, is that establishing a shared secret key and keeping it secret may be difficult.

The symmetric cryptography scheme that will be used in the Onion Routing implementation described in this report is called Advanced Encryption Standard (AES) [6]. The 128-bit key variant is used for ease of development, especially in the testing phase, but the `stealth` library developed for this implementation provides accomodations for easily changing the key size to 192- or 256-bit keys. It is generally frowned-upon to manually implement cryptographic schemes, as it is always preferable to use libraries that have been thoroughly tested and validated. The PyCrypto library was used to provide primitive AES encryption and decryption, as well as cryptographically-secure random key generation, in this implementation.

A.3 Asymmetric Cryptography

Although symmetric cryptography certainly has its benefits, it requires two parties to share a secret key, which may not be easily done, especially if the key must be communicated over a network. Asymmetric cryptography solves this issue, though it has issues of its own which will be described soon.

Instead of two parties sharing a shared secret key as in symmetric cryptography, asymmetric cryptography schemes give each party both a *private key* and a *public key*. As the names suggest, the public key can be known by anyone, but the private key should be kept secret. Note however that although the private key must remain secret, it is not shared by anyone, which removes the main drawback of symmetric encryption. Like symmetric encryption, asymmetric encryption schemes are defined by $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$, with $\mathsf{Enc}_{pub} : \mathcal{M} \to \mathcal{C}$ and $\mathsf{Dec}_{priv} : \mathcal{C} \to \mathcal{M}$. The encryption/decryption process has the following property:

$$\mathsf{Dec}_{f(k)}(\mathsf{Enc}_k(m)) = m \tag{A.2}$$

In equation (A.2), k is the public key, and $f(k)$ yields the private key. Asymmetric cryptography is only effective if computing $f(k)$ is an NP problem – that is to say, computing $f(k)$ is computationally infeasible. Given this property, if user $\mathcal{A}$ wants to send an encrypted message to $\mathcal{B}$, user $\mathcal{A}$ encrypts the message with user $\mathcal{B}$'s public key (which is publicly-known), and then $\mathcal{B}$ may decrypt the message using its private key. Since $f(k)$ is computationally intractible, the probability of someone other than $\mathcal{B}$ decrypting the message is negligible.

In the implementation of Onion Routing discussed in this report, asymmetric cryptography is used to communicate symmetric keys. Since public keys are publicly-known, the sender knows how to send encrypted messages to any onion router without needing a shared secret key. Therefore, asymmetric cryptography allows the sender to share symmetric keys without anyone aside from the desired router being able to see the symmetric key.

Although asymmetric cryptography does provide some incredible benefits, it also has its drawbacks. Firstly, asymmetric encryption and decryption is relatively slow [5], so it is beneficial to reduce the frequency of its use as much as possible without comprimising the security of the system. Furthermore, in some applications, such as the main purpose of onion routers, for example, it may be necessary to send encrypted *and* decrypted messages over the network. In this scenario, asymmetric cryptography cannot be used. In the case of the onion routers, they must encrypt response messages before forwarding them back towards the sender. Since the sender does not know the onion routers' private keys, the sender cannot decrypt the response! Therefore, clearly asymmetric encryption cannot be used in some scenarios.

The cryptographic scheme that is used in the Onion Routing implementation presented in this report is called RSA [7]. As in the case of symmetric cryptography, it is always preferred to use trusted libraries for implementing asymmetric cryptography. The PyCrypto library also implements the RSA cryptosystem, and it was used in the implementation described in this report.

B. List of External Resources

Several external, third party resources were used in the implementation described in this report. Below is a list of these resources, as well as references to where they can be found/downloaded and descriptions of what they were used for.

Resource	Origin	Reason for Use
PyCrypto python library	https://www.dlitz.net/software/pycrypto/	AES encryption/decryption, RSA encryption/decryption, importing and exporting of RSA keys, random byte string generation, pseudorandom number generators

Bibliography

[1] J. Camenisch and A. Lysyanskaya, "A formal treatment of onion routing," in *Annual International Cryptology Conference*, pp. 169–187, Springer, 2005.

[2] M. G. Reed, P. F. Syverson, and D. M. Goldschlag, "Onion routing network for securely moving data through communication networks," July 24 2001. US Patent 6,266,704.

[3] D. M. Goldschlag, M. G. Reed, and P. F. Syverson, "Hiding routing information," in *International Workshop on Information Hiding*, pp. 137–150, Springer, 1996.

[4] J. Niehren, J. Schwinghammer, and G. Smolka, "A concurrent lambda calculus with futures," *Theoretical Computer Science*, vol. 364, no. 3, pp. 338–356, 2006.

[5] M. Agrawal and P. Mishra, "A comparative survey on symmetric key encryption techniques," *International Journal on Computer Science and Engineering*, vol. 4, no. 5, p. 877, 2012.

[6] J. Daemen and V. Rijmen, "Announcing the advanced encryption standard (aes)," *Federal Information Processing Standards Publication*, vol. 197, 2001.

[7] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.