ENHANCING THE BOOTSTRAPPING NETWORK
IN CLOUD-BASED ONION ROUTING (COR)

by

KEVIN SNIDER

B.A., Dartmouth College, 2010

A thesis submitted to the Graduate Faculty of the
University of Colorado Colorado Springs
in partial fulfillment of the requirements for the degree of
Master of Science
Department of Computer Science
2017
This thesis for the Master of Science degree by

Kevin Snider

has been approved for the Department of Computer Science

by

Edward Chow, Chair

Sang-Yoon Chang

Yanyan Zhuang

________12/12/17________

Date
Snider, Kevin (M.S. Computer Science)

Enhancing the Bootstrapping Network in Cloud-based Onion Routing (COR)

Thesis directed by Professor Edward Chow.

Abstract

The maintenance of privacy and anonymity on the Internet has become a large concern for many users worldwide. The online service known as Tor (The Onion Router) is one of the most popular tools available to users seeking to preserve their privacy and anonymity when browsing online or accessing the Internet in general. This paper describes my efforts into developing an implementation of a bootstrapping network for cloud-based onion routing (COR), an idea which seeks to adapt the concepts behind the Tor service by applying them to a cloud-services environment. The preliminary results show an anonymous connection can be built and retrieve the desired payload in an average response time of 5 seconds for a 4-hop circuit, expanding up to 20 seconds for a 10-hop circuit.
# TABLE OF CONTENTS

CHAPTER

I. INTRODUCTION ........................................................................................................... 1

II. RELATED WORK ......................................................................................................... 6

III. THESIS GOAL ............................................................................................................. 9

IV. DESIGN ...................................................................................................................... 11

V. IMPLEMENTATION ................................................................................................... 15

  5.1. ASP Implementation .............................................................................................. 15

  5.2. Relay Implementation ........................................................................................... 16

  5.3. Client Implementation .......................................................................................... 16

VI. EVALUATION ........................................................................................................... 18

VII. LESSONS LEARNED .............................................................................................. 23

VIII. FUTURE DIRECTIONS ......................................................................................... 25

IX. CONCLUSION ........................................................................................................... 27

REFERENCES ................................................................................................................. 28

APPENDIX A: INSTALLATION AND CONFIGURATION ............................................. 29

  A.1. Library Installation ............................................................................................... 29

  A.2. Source File Distribution ...................................................................................... 30

  A.3. System Configuration .......................................................................................... 31

APPENDIX B: DEMONSTRATION ............................................................................... 36
LIST OF FIGURES

FIGURE

1. The onion layering of messages, with destination and payload encrypted by the public key of the corresponding relay (or router). ................................................................. 2

2. The onion routing network, with emphasis on the bootstrapping network portion. ...... 3

3. A bootstrapping network implemented using cloud service providers. ................. 4

4. Diagram of steps involved in COR Bootstrap Network design. .............................. 11

5. Connection Latency by Circuit Length for Different Buffer Sizes ....................... 21

6. Demo Sequence No. 1. ......................................................................................... 37

7. Demo Sequence No. 2. ......................................................................................... 38

8. Demo Sequence No. 3. ......................................................................................... 39

9. Demo Sequence No. 4. ......................................................................................... 40

10. Demo Sequence No. 5. ....................................................................................... 41

11. Demo Sequence No. 6. ....................................................................................... 42

12. Demo Sequence No. 7. ....................................................................................... 43

13. Demo Sequence No. 8. ....................................................................................... 44

14. Demo Sequence No. 9. ....................................................................................... 45

15. Demo Sequence No. 10. ..................................................................................... 46

16. Demo Sequence No. 11. ..................................................................................... 47

17. Demo Sequence No. 12. ..................................................................................... 48

18. Demo Sequence No. 13. ..................................................................................... 49

19. Demo Sequence No. 14. ..................................................................................... 50

20. Demo Sequence No. 15. ..................................................................................... 51
<table>
<thead>
<tr>
<th>Demo Sequence No.</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>52</td>
</tr>
<tr>
<td>17</td>
<td>53</td>
</tr>
<tr>
<td>18</td>
<td>54</td>
</tr>
<tr>
<td>19</td>
<td>55</td>
</tr>
<tr>
<td>20</td>
<td>56</td>
</tr>
<tr>
<td>21</td>
<td>57</td>
</tr>
<tr>
<td>22</td>
<td>58</td>
</tr>
<tr>
<td>23</td>
<td>59</td>
</tr>
<tr>
<td>24</td>
<td>60</td>
</tr>
<tr>
<td>25</td>
<td>61</td>
</tr>
<tr>
<td>26</td>
<td>62</td>
</tr>
<tr>
<td>27</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>
LIST OF TABLES

TABLE

1. Latency measurements (seconds) for buffer size 2048 ................................................. 18
2. Latency measurements (seconds) for buffer size 4096 .................................................. 19
3. Latency measurements (seconds) for buffer size 8192 .................................................. 19
4. Latency measurements (seconds) for buffer size 16384 .............................................. 20
5. Latency measurements (seconds) for buffer size 32768 .............................................. 20
CHAPTER I
INTRODUCTION

The problem of maintaining one's privacy and anonymity on the Internet has become one of the major issues at the forefront of many users' concerns. Many users find themselves uncomfortable at the prospect of their online actions being tracked, whether it be to serve corporate interests, the actions of malicious third parties, or as a function of government surveillance and/or censorship. This paper describes my efforts into developing a tool to counteract these privacy-degrading forces: an implementation of a bootstrapping network for cloud-based onion routing (COR), an idea which seeks to apply the concept behind the popular privacy-preserving service known as Tor (The Onion Router) to a cloud-services environment.

The Tor network began operating in 2003 and is designed for preserving user anonymity, and has now become one of the most popular tools for accessing the Internet anonymously. Tor functions by tunneling a user’s traffic through a series of proxies or "relays", with the effect that the traffic appears to have originated from the last relay in the circuit, rather than the original user.
Figure 1. The onion layering of messages, with destination and payload encrypted by the public key of the corresponding relay (or router).

To conceal the sender’s identity, the sender encrypts the destination and the message to be sent by the last relay to the destination with the public key of the last relay, also known as the exit router. Therefore, no other relays can know the message content and the final destination. The sender then takes this encrypted message as the payload, prefixes it with the domain name or IP address of the relay which is second from the last in the circuit, then encrypts the result with the public key of that second-to-last relay. This encapsulation and encryption process is repeated for each relay node along the selected path until the first relay, also called the entry node, is reached. This final resulting message is sent by the sender to the entry node. Each relay will be able to decrypt the message and reveal the next node in the circuit to send the next encrypted message to, in a way "peeling" away the wrapped layered message like the layers in an onion - hence the name "onion routing".
Figure 2 shows the architecture of the onion network, where the Tor relays are hosted by volunteers throughout the world. To set up an onion circuit, the client first sets up an anonymous connection to a Tor directory server via the bootstrapping network, which serves to hide the identity of the client. The client then requests the router list from a set of these Tor directory servers. Finally, the client chooses a subset of routers from the returned list to form the Tor circuit used in the encapsulation process depicted in Figure 1.
An example of just such a bootstrapping network can be seen above in Figure 3. Here we show three Anonymity Service Providers (ASP) providing their anonymity services in two different clouds. Each ASP maintains a set of virtual machines in each cloud to be used as relays, and must be assumed not to collaborate with each other. To use the ASP service, a client sends a request with a proposed login and password to an ASP, say ASP1. ASP1 then creates an account on a set of virtual machines in different clouds, then sends back the set to the client. The client then picks a virtual machine, say R1, from one of the clouds and logs in there (shown in Figure 3 as the red VM in Cloud 1). From R1, the client repeats the process by requesting a set of virtual machines from ASP2. Assume it picks the G2 virtual machine from this set. The client then sends a request from G2 to ASP3, and chooses B1 as the last node in the circuit. From B1, the client can query any of the Tor directory servers for the list of Tor routers. Note that by
zigzagging back and forth between the two clouds we greatly increase the difficulty of tracing the connection back to the actual client, serving to obscure the client's identity.
CHAPTER II
RELATED WORK

Reed et al [Reed1998] first introduced the concept of onion routing in 1998 out of the Naval Research Laboratory. In their paper, they outline the basic process of encapsulating the routing information of an "onion" packet in layers and how it may be directed through entry, middle, and exit servers in an onion network, each server stripping off the onion's outer layer for the routing and encryption data relevant only to their next hop in the network.

Dingledine et al [Ding2004] present the seminal paper on Tor, the most well-known and widely used implementation of an onion-routing network. They outline their design goals and intended usage of Tor, provide a detailed explanation of the protocol functionality and server-side policies, and conclude with an evaluation of Tor's perceived effectiveness against various possible attacks.

McCoy et al [McCo2008] conduct a practical review of Tor, evaluating whether the intended goals of Tor are being met in its actual implementation. They analyze the different types of web traffic Tor is used for, as well as the geopolitical distribution of both users and hosts in the Tor network, and compare these with the theoretical ideals of the Tor design. They also note real-world instances of misuse by both users and hosts, whether malicious or simply negligent, that can also compromise a user's anonymity.

Khattak et al [Khat2016] present a wider view of censorship resistance systems as a whole, analyzing their common goals and purpose, while comparing their different strategies and their related strengths/weaknesses. They seek to create a categorization of
these different systems to more cleanly and efficiently identify responses to a variety of possible censor attacks.

Tschantz et al [Tsch2016] take the analysis of censorship resistance systems a step further, seeking to ground their analysis empirically against real-world censorship. They encompass a wide group of actual censorship resistance systems, notate each for the particular strategies they employ, and analyze their performance against different nation-state censors such as China and Iran. They also inform their analysis by taking the opposing viewpoint - namely, how are the censors going about enacting their censorship and what are their decisions on the cost/benefit relationship - and attempt to use this understanding of the censors' goals against them.

Lincoln et al [Linc2012] identify the bootstrapping network necessary to initiate the connection into the Tor network as a particular point of vulnerability, and discuss a method to enhance this step in the process. Their DEFIANCE framework serves as an extension to the Tor network and presents a possible strategy to improve the resilience of bootstrapping into the network's entry node.

Brubaker et al [Brub2014] propose the idea of using cloud storage to serve as nodes in a censorship resistance system, in order to decrease cost as well as further frustrate censors. Their rationale is that once a Tor node is identified, it is easy for a censor to block traffic from that node using filtering attacks. If the node is placed on a cloud storage server, however, the censor becomes much less likely to block that node's IP because the virtual server can easily be taken down and replaced. The censor would be forced to block all IPs from that cloud provider, an action they are unlikely to take because of the unacceptable collateral damage incurred.
Jones et al [Jone2011] outline a practical implementation of a cloud-based onion routing (COR) network. They describe how the goals of both the bootstrapping network and the data network used for communicating can be realized using a cloud storage infrastructure. In addition, they provide a detailed explanation of their protocol and intended usage guidelines, and supplement these instructions with performance evaluations of their design under different circumstances.
CHAPTER III
THESIS GOAL

The goal of this thesis is to design and implement an efficient bootstrapping network framework which allows an ASP to provide anonymity services on a cloud-based architecture, and provides to a client the ability to set up a secure bootstrapping circuit that protects their privacy.

The tasks for this thesis project are as follows:

1. Create virtual machines on different Amazon Web Services (AWS) regions, to simulate different cloud providers. Some of them will form the ASP providers, and others the relay virtual machines the ASPs will manage.

2. Create the ASP service on the ASP virtual machines. This will involve a front-end web page to accept requests from a client and server-side scripts to register connection-specific credentials on them, distribute these credentials to their designated relays, and return the available set of relays back to the client.

3. Develop a software tool to allow a client to specify the desired circuit length with the number of clouds to hop back-and-forth between in the circuit, then to actually realize the specified circuit by requesting the corresponding ASPs and setting up the connections through the circuit. The result will be a connection where the client can issue a request to the Tor network or perform some other privacy-preserving purpose.

4. Develop the relay service software to integrate between the client and ASPs in order to perform the incremental layer-by-layer onion routing necessary to create the aforementioned circuit.
5. Document the design and implementation of the bootstrapping network prototype. Perform a performance evaluation of the baseline bootstrapping network and propose alternate design and/or implementation suggestions for comparison.
I designed my interpretation of the COR Bootstrapping Network to function as follows (refer to corresponding Figure 4 for diagram depicting these steps):

1. User enters circuit length (assume 2 for this example), filename containing ASP addresses (asps.txt), and address of payload web page to retrieve into client program.
2. Client reads in asps.txt from file.
3. Client parses asps.txt and randomly selects first ASP to connect to (assume ASP1), then generates random login, password, and key (L1, P1, and K1) for first hop in circuit.
4. Client submits L1, P1, and K1 to the Input Page on ASP1 over https.
5. Input Page on ASP1 calls script to store L1, P1, and K1 to credentials database on ASP1, and all relays managed by ASP1 via SSH. (Only “Relay 1” depicted, but assumed to send to all others.)

6. ASP1 returns its Relay List to client.

7. Client parses Relay List and selects one at random (assume it selects “Relay 1”). Then client creates message to send to Relay 1: Because the client wants another hop, the Command will be “Extend”, followed by the address of a different ASP to connect to (assume ASP2), and a newly generated set of login, password, and key (L2, P2, and K2) to submit to ASP2 for second hop in circuit. Then client prepends P1 to this message, encrypts the message with K1, then prepends L1 to the result.

8. Client sends this message to relay program running on Relay 1.

9. Relay on Relay 1 strips off L1 from the message and retrieves the entry with that login from its credentials database. If no entry found, connection is broken.

10. Relay on Relay 1 decrypts remaining message with corresponding K1 from database entry, then strips off P1 from message and compares it to P1 from database entry. If they don’t match, connection is broken.

11. If they do match, relay forks off child process to continue listening for connections while this connection is handled.

12. Relay on Relay 1 reads Command in remaining message, sees that it is “Extend”. It then interprets the remaining message as the address of the ASP to connect to and the login, password, and key to submit to it. (Here ASP2 and L2, P2, and K2).

13. Relay on Relay 1 submits L2, P2, and K2 to the Input Page on ASP2 over https.
14. Input Page on ASP2 calls script to store L2, P2, and K2 to credentials database on ASP2, and all relays managed by ASP2 via SSH. (Only “Relay 2” depicted, but assumed to send to all others.)

15. ASP2 returns its Relay List to relay on Relay 1.

16. Relay on Relay 1 creates message by encrypting Relay List with K1.

17. Relay on Relay 1 sends message back to client.

18. Client decrypts result using K1, then parses the resulting message containing the Relay List from ASP2 and selects one at random (assume it selects “Relay 2”). Then client creates message to send to Relay 2 through Relay 1: Because the client does not want any more hops beyond Relay 2, the Command will be “Get”, followed by the address of the payload web page. Then client prepends P2 to this message, encrypts it with K2, then prepends L2. Because this message is going through Relay 1, the client prepends the Command “Transfer” and the address of Relay 2 to the message. Then client prepends P1, encrypts it with K1, and prepends L1.

19. Client sends this message to relay on Relay 1.

20. Relay on Relay 1 strips off L1 from the message and decrypts remaining message using previously retrieved K1, then strips off P1. If either L1 or P1 don’t match, connection is broken. Otherwise, relay on Relay 1 strips off the Command in remaining message, sees that it is “Transfer”, and interprets it to strip off the address from the message (address of “Relay 2”).

21. Relay on Relay 1 sends remaining message to relay on Relay 2.

22. Relay on Relay 2 strips off L2 from the message and retrieves the entry with that login from its credentials database. If no entry found, connection is broken.
23. Relay on Relay 2 decrypts remaining message with corresponding K2 from database entry, then strips off P2 from message and compares it to P2 from database entry. If they don’t match, connection is broken.

24. If they do match, relay forks off child process to continue listening for connections while this connection is handled.

25. Relay on Relay 2 reads Command in remaining message, sees that it is “Get”. It then interprets the remaining message as the address of the payload web page to retrieve.

26. Relay on Relay 2 sends request to specified address of payload web page.

27. Web provider returns requested payload to relay on Relay 2.

28. Relay on Relay 2 creates message by encrypting payload with K2.

29. Relay on Relay 2 sends message back to relay on Relay 1.

30. Relay on Relay 1 encrypts message with K1.

31. Relay on Relay 1 sends this message back to client.

32. Client decrypts message using K1, then decrypts result using K2.

33. Client returns remaining message consisting of final payload to the user.
To implement this proposed design, I turned to Amazon Web Services (AWS) to provide my cloud-computing needs. To represent the distinct and separate clouds called for in the bootstrapping protocol, I designated three separate regions within the AWS EC2 service: us-west-2 (Oregon), us-east-2 (Ohio), and us-east-1 (Virginia). In total, I used 13 Amazon Linux AMI t2.micro virtual machine instances, each with an allocated size of 8 GB and an associated elastic IP address. The breakdown of these 13 instances is as follows:

- 3 ASP servers to maintain relays across the separate regions, simply referred to as asp1, asp2, and asp3. Region is irrelevant for ASP functionality, but I assigned asp1 to Oregon, asp2 to Ohio, and asp3 to Virginia in order to simulate redundancy across separate sites for the ASP service.
- 9 relays (3 in each region, with one in each region belonging to each ASP). For example, the 3 relays in Oregon were dubbed asp1-or, asp2-or, and asp3-or, the prefix in each name relating to which ASP the relay belongs to. Likewise, the 3 in Ohio were named asp1-oh, asp2-oh, and asp3-oh, and the 3 in Virginia asp1-va, asp2-va, and asp3-va.
- 1 instance to simulate a client accessing the network, randomly created in the Ohio region, but could just as easily have been assigned to any other region.

5.1. ASP Implementation

The ASP service consisted of an HTTPS-enabled Apache web server, presenting a simple HTML input page to users navigating to the ASP's address, with input forms to
accept values for a login, password, and key to be associated with a connection request. These inputs would be processed by a PHP script which would store the input credentials to a local MySQL database instance as well as distribute them via SSH to MySQL database instances on the relays maintained by the ASP in question (each ASP also maintains a database table consisting of the addresses of the relays belonging to it). Once this task was completed, the script would return a page to the user consisting of this list of relays that belonged to that particular ASP.

5.2. Relay Implementation

The relay instances utilized the same Apache-MySQL-PHP setup as the ASPs on the webserver end to distribute credential requests for connections, the difference being that the relays would receive their inputs as automated requests from the ASP services, and would simply store the inputs to their own local databases without distributing them, as unlike the ASPs the relays would have no database table listing the relays maintained.

Each relay also runs a copy of a highly-complicated C socket program that is responsible for the tasks of listening for and accepting network socket connections, validating and decrypting incoming messages, interpreting and executing different routing commands, and repackaging response messages to return to the requesting source. This program utilized TCP stream socket protocol for network protocols and 128-bit AES encryption routines.

5.3. Client Implementation

Unlike the other instances, the client runs no webserver functions, as they are not providing the service, merely accessing it. The accessing of the service does require some special functionality on the client end as well - they will need have access to a
separate client-side C socket program, as well as a list of available ASPs to provide as 
input to the client program. This client program manages the tasks of accepting the user-
input destination address to retrieve, the desired circuit length, and the ASP list, then 
randomly generating and registering credentials to selected ASPs, packaging credentials 
and commands into repeated onion layers for each step, connecting through selected 
relays to progressively build the circuit, and receiving the final payload from the 
destination address to decrypt and display to the user.
CHAPTER VI
EVALUATION

To evaluate the performance of the bootstrapping network prototype, I measured the connection latency between the client command and the delivery of the desired payload at each circuit length up to 10 (with a circuit length of 0 representing a direct connection involving no circuit building or onion-routing). In order to accommodate for variance between connection requests, I ran 10 trials at each circuit length and took the average of those 10 trials. To evaluate the impact of the size of the message length buffer, I repeated these same measurements for message buffer sizes of 2048, 4096, 8192, 16384, and 32768 bytes. Below are the data sets for each buffer size:

Table 1: Latency measurements (seconds) for buffer size 2048

<table>
<thead>
<tr>
<th>Circuit Length</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial 1</td>
<td>0.147</td>
<td>1.892</td>
<td>2.975</td>
<td>5.050</td>
<td>4.238</td>
<td>7.603</td>
<td>8.053</td>
<td>16.468</td>
<td>14.341</td>
<td>14.067</td>
<td>23.756</td>
</tr>
<tr>
<td>Trial 4</td>
<td>0.147</td>
<td>1.282</td>
<td>2.974</td>
<td>3.471</td>
<td>4.093</td>
<td>5.288</td>
<td>7.880</td>
<td>11.587</td>
<td>16.091</td>
<td>17.717</td>
<td>27.401</td>
</tr>
<tr>
<td>Trial 5</td>
<td>0.151</td>
<td>0.987</td>
<td>1.852</td>
<td>3.261</td>
<td>5.227</td>
<td>6.854</td>
<td>10.028</td>
<td>11.937</td>
<td>10.538</td>
<td>18.620</td>
<td>14.674</td>
</tr>
<tr>
<td>Trial 7</td>
<td>0.149</td>
<td>1.126</td>
<td>1.823</td>
<td>3.816</td>
<td>6.177</td>
<td>5.413</td>
<td>11.065</td>
<td>10.625</td>
<td>9.940</td>
<td>16.892</td>
<td>15.401</td>
</tr>
<tr>
<td>Trial 8</td>
<td>0.152</td>
<td>1.101</td>
<td>2.562</td>
<td>3.152</td>
<td>4.182</td>
<td>5.582</td>
<td>12.024</td>
<td>12.301</td>
<td>15.368</td>
<td>17.082</td>
<td>13.891</td>
</tr>
<tr>
<td>Trial 10</td>
<td>0.150</td>
<td>1.050</td>
<td>1.677</td>
<td>3.995</td>
<td>5.879</td>
<td>7.613</td>
<td>7.003</td>
<td>7.108</td>
<td>12.303</td>
<td>13.994</td>
<td>28.147</td>
</tr>
<tr>
<td>Average</td>
<td><strong>0.162</strong></td>
<td><strong>1.229</strong></td>
<td><strong>2.175</strong></td>
<td><strong>3.536</strong></td>
<td><strong>4.947</strong></td>
<td><strong>6.383</strong></td>
<td><strong>9.077</strong></td>
<td><strong>10.835</strong></td>
<td><strong>13.504</strong></td>
<td><strong>17.582</strong></td>
<td><strong>19.839</strong></td>
</tr>
</tbody>
</table>
Table 2: Latency measurements (seconds) for buffer size 4096

| Circuit Length | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  |
|----------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Trial 1        | 0.273 | 1.546 | 2.358 | 3.552 | 4.897 | 8.277 | 10.155 | 8.446 | 14.944 | 18.130 | 18.172 |
| Trial 4        | 0.147 | 0.700 | 2.462 | 2.765 | 4.821 | 8.667 | 10.228 | 13.391 | 12.706 | 16.673 | 19.972 |
| Trial 5        | 0.144 | 0.698 | 1.967 | 3.548 | 5.770 | 7.353 | 8.390 | 10.826 | 14.312 | 16.362 | 19.767 |
| Trial 8        | 0.148 | 1.118 | 2.426 | 4.396 | 5.141 | 5.519 | 6.184 | 12.827 | 17.205 |
| Trial 10       | 0.144 | 1.195 | 2.025 | 3.233 | 4.545 | 7.125 | 9.421 | 15.834 | 17.405 |
| **Average**    | 0.166 | 1.145 | 2.164 | 3.397 | 4.966 | 7.086 | 9.435 | 10.934 | 14.184 |

Table 3: Latency measurements (seconds) for buffer size 8192

| Circuit Length | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  |
|----------------|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| Trial 1        | 0.175 | 0.910 | 1.843 | 3.143 | 5.659 | 5.344 | 8.016 | 8.797 | 17.583 | 18.232 | 28.459 |
| Trial 4        | 0.152 | 0.846 | 2.632 | 2.999 | 6.047 | 5.860 | 7.798 | 11.442 | 17.828 | 13.429 | 19.854 |
| Trial 5        | 0.152 | 0.682 | 1.745 | 3.665 | 4.588 | 5.768 | 10.603 | 10.325 | 12.908 | 19.459 | 25.459 |
| Trial 6        | 0.151 | 0.831 | 1.929 | 2.969 | 4.247 | 6.290 | 11.785 | 13.371 | 15.105 | 13.799 | 15.039 |
| Trial 7        | 0.149 | 0.707 | 1.623 | 3.274 | 4.243 | 5.885 | 9.147 | 16.022 | 14.362 | 24.812 | 18.797 |
| Trial 8        | 0.144 | 0.693 | 1.646 | 3.737 | 4.098 | 5.546 | 7.151 | 13.419 | 13.046 | 17.655 | 14.377 |
| Trial 9        | 0.168 | 0.988 | 1.740 | 3.129 | 6.155 | 7.776 | 7.201 | 9.297 | 14.046 | 17.948 | 22.160 |
Table 4: Latency measurements (seconds) for buffer size 16384

<table>
<thead>
<tr>
<th>Circuit Length</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial 1</td>
<td>0.173</td>
<td>0.878</td>
<td>1.835</td>
<td>2.722</td>
<td>4.813</td>
<td>5.210</td>
<td>7.292</td>
<td>14.417</td>
<td>16.974</td>
<td>15.756</td>
<td>15.970</td>
</tr>
<tr>
<td>Trial 2</td>
<td>0.154</td>
<td>0.974</td>
<td>1.972</td>
<td>3.949</td>
<td>5.285</td>
<td>8.010</td>
<td>12.059</td>
<td>12.718</td>
<td>10.663</td>
<td>26.331</td>
<td>19.968</td>
</tr>
<tr>
<td>Trial 3</td>
<td>0.147</td>
<td>0.863</td>
<td>1.818</td>
<td>3.712</td>
<td>4.466</td>
<td>7.993</td>
<td>10.426</td>
<td>11.731</td>
<td>17.900</td>
<td>22.468</td>
<td>24.074</td>
</tr>
<tr>
<td>Trial 5</td>
<td>0.148</td>
<td>1.198</td>
<td>2.594</td>
<td>3.005</td>
<td>6.801</td>
<td>5.724</td>
<td>10.627</td>
<td>13.016</td>
<td>11.362</td>
<td>18.770</td>
<td>17.622</td>
</tr>
<tr>
<td>Trial 6</td>
<td>0.169</td>
<td>0.763</td>
<td>1.813</td>
<td>3.019</td>
<td>5.882</td>
<td>6.525</td>
<td>9.530</td>
<td>8.857</td>
<td>10.939</td>
<td>16.329</td>
<td>18.757</td>
</tr>
<tr>
<td>Trial 10</td>
<td>0.148</td>
<td>0.912</td>
<td>1.832</td>
<td>2.971</td>
<td>4.706</td>
<td>8.317</td>
<td>10.234</td>
<td>10.213</td>
<td>16.234</td>
<td>12.686</td>
<td>22.896</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>0.166</td>
<td>1.048</td>
<td>2.067</td>
<td>3.287</td>
<td>5.004</td>
<td>7.091</td>
<td>9.877</td>
<td>11.165</td>
<td>13.590</td>
<td>17.722</td>
<td>20.121</td>
</tr>
</tbody>
</table>

Table 5: Latency measurements (seconds) for buffer size 32768

<table>
<thead>
<tr>
<th>Circuit Length</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial 1</td>
<td>0.201</td>
<td>1.541</td>
<td>2.102</td>
<td>2.763</td>
<td>3.899</td>
<td>8.792</td>
<td>10.262</td>
<td>10.499</td>
<td>18.376</td>
<td>15.886</td>
<td>15.687</td>
</tr>
<tr>
<td>Trial 3</td>
<td>0.148</td>
<td>1.020</td>
<td>1.859</td>
<td>3.027</td>
<td>5.948</td>
<td>5.825</td>
<td>7.543</td>
<td>11.252</td>
<td>16.183</td>
<td>17.793</td>
<td>17.412</td>
</tr>
<tr>
<td>Trial 5</td>
<td>0.147</td>
<td>0.705</td>
<td>1.862</td>
<td>4.067</td>
<td>4.211</td>
<td>6.245</td>
<td>10.825</td>
<td>11.300</td>
<td>14.221</td>
<td>20.242</td>
<td>21.122</td>
</tr>
<tr>
<td>Trial 6</td>
<td>0.152</td>
<td>0.708</td>
<td>1.980</td>
<td>3.477</td>
<td>4.174</td>
<td>7.801</td>
<td>8.009</td>
<td>12.119</td>
<td>11.265</td>
<td>17.598</td>
<td>17.327</td>
</tr>
<tr>
<td>Trial 8</td>
<td>0.152</td>
<td>0.994</td>
<td>2.110</td>
<td>2.761</td>
<td>5.425</td>
<td>8.072</td>
<td>9.317</td>
<td>14.089</td>
<td>17.990</td>
<td>20.845</td>
<td>21.068</td>
</tr>
<tr>
<td>Trial 9</td>
<td>0.152</td>
<td>0.861</td>
<td>1.962</td>
<td>3.695</td>
<td>6.312</td>
<td>5.769</td>
<td>9.151</td>
<td>12.843</td>
<td>13.110</td>
<td>19.736</td>
<td>24.296</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>0.168</td>
<td>1.094</td>
<td>1.982</td>
<td>3.287</td>
<td>5.124</td>
<td>7.026</td>
<td>8.775</td>
<td>11.645</td>
<td>15.286</td>
<td>17.648</td>
<td>20.129</td>
</tr>
</tbody>
</table>
To better visualize these results, I have collated them into the below graph, depicting the average connection latency at each circuit length for each message buffer size.

![Graph showing connection latency by circuit length for different buffer sizes.](image)

**Figure 5. Connection Latency by Circuit Length for Different Buffer Sizes**

As can be seen here, the driving factor in latency is the circuit length, exhibiting a slight exponential curve in latency as circuit length increases. We see a near 0 latency for a direct connection (circuit length 0), increasing to about 1 second latency for a one-hop circuit. This increases to approximately a 5-second latency for a circuit length of 4, eventually reaching about 20 seconds at a circuit length of 10. While a 20-second latency may sound like an unreasonable delay, it must be noted that this is at the extreme end of a 10-hop circuit, which would signify a user obsessively concerned with privacy. In this
regard, the sacrifice of a longer delay must be considered from the perspective of a trade-off between security and efficiency. A strong level of anonymity may still be obtained by a more reasonable 4-hop circuit, for which a mere 5 second delay should not be overly oppressive to the user.

Interestingly, the graph depicts how there is not that strong of an effect attributed to the message buffer size. We can see the latency measurements begin to expand out and differentiate from one another more as we reach the larger circuit lengths, but with no immediately obvious trend to describe any particular causal relationship. I take this observation to signify that the driving factor in determining the latency is the logic steps taken within the software between each transmission from node to node. Alternatively, another explanation could be that because the distances traveled from node-to-node are so great in making hops across the country, all other effects are minimized as to be comparatively trivial.
CHAPTER VII
LESSONS LEARNED

The lessons I learned over the course of this project were numerous and varied. The first one I learned was actually one I thought I already knew, but still managed to give myself a hands-on lesson in, which is the importance of laying out a thorough groundwork of design before embarking on any software-related task. Eager to jump in and make headway on the project, I began writing socket routines to send messages back-and-forth between different nodes, then expanded from there until I had created my first iteration of the project, which involved Diffie-Helman style RSA-encrypted key exchanges between each node prior to every connection, and, while it had the basics of routing encrypted traffic through multiple relays, was woefully inefficient and an incorrect application of onion-routing principles. It also did not feature any sort of credential-distribution mechanism, and thus failed in measures of the security principle of integrity. While I had learned about and given lip-service to the importance of design in previous classes, the act of essentially scrapping my first iteration and rebuilding from scratch based on my new, complete design served as a heavy re-emphasis on the importance of having a fully thought out plan in advance.

That is not to say that there were not other lessons learned during this first phase as well. Indeed, having never written a C socket program before, the process of creating a simple server to listen for and accept connection requests was very illuminating to me. I came to be much more comfortable with the process, but at the outset the understanding of the syntax of the various network system calls was frustrating, as well as the matching
of send / receive protocols on each end and making sure the correct number of bytes were sent and received on both ends.

I learned how to write a C routine to encrypt data using RSA encryption, then learned that RSA was about 10 times slower than AES encryption and inappropriate for the purposes of the network traffic of my design. Therefore, I learned about the correct application of AES encryption, and how to write a C routine to encrypt data using AES encryption.

The introduction of the webserver and HTTPS credential-distribution functionality also presented a whole new slew of learning opportunities for me, as these activities were outside my realm of experience as well. During this stage of implementation, I learned the following skills

- Apache httpd service installation and configuration
- MySQL service installation and configuration, basic database management commands
- expanded HTML webpage programming skills
- proper use and reasons for Base64 and URL encoding
- PHP scripting, including Base-64 encoding, Curl, MySQL, etc. commands
- HTTPS protocols, key management schemes
CHAPTER VIII
FUTURE DIRECTIONS

What seemed a simple concept at the outset rapidly expanded in scope to a much larger project than I had originally anticipated. While the presented implementation of the COR Bootstrap Network represents a complete, working prototype, there are a number of refinements that I have already identified that could be made to further improve upon the design.

The first and most obvious is the need to send the entire buffer size through each node in the circuit all at once, rather than in multiple packets of the content broken down into smaller chunks. To do this would require further complex logic in handling out-of-order arrival of separate packages at different stages in the circuit, but could potentially cut down on the large encryption/decryption and transmission times required for such a large block of data.

Another refinement that could be made is to allow the relay service to run across a larger range of ports. As it stands right now, each relay is hard-coded to only listen for connections on port 9500, which was sufficient for my purposes to enable socket connections from node to node to demonstrate the circuit-building functionality within COR. Greater efficiency could be achieved for such a system on a larger scale if each relay managed connections across multiple ports, thus limiting the congestion that can result from a larger number of users bottlenecking through a single port.

Another feature that could result in greater efficiency of this system on a larger scale would be to perform the credentials validation before forking off to handle the connection, so that in the event of an invalid connection being discarded, the expensive
forking operation could be avoided. I could not assess the impact of such a change on the scale of my simple prototype, as all the connections I sent through the bootstrap network were intended to be valid connections. In the event of a larger-scale deployment of this system, such an addition could theoretically be of much benefit, especially under the potential circumstances of a man-in-the-middle or DDoS attack.

Another potential source of greater efficiency lies within the function that iteratively builds the layers of the message. To create the effect of prepending headers after each stage of message creation, I found it necessary to utilize a number of memmove function calls to open up space in the message buffer for the header to be placed. While the function does perform its purpose, I feel that great gains in speed could be made were a more efficient approach utilized.

Although I feel they were not necessary at this stage, a couple of minor cosmetic improvements could also be applied. Currently the client program returns the final payload as the bare HTML text contents to the user, rather than the formatted webpage output the layperson would expect. In addition, there is no functionality to interact with any webpage accessed, as the bootstrap network currently serves only to retrieve the contents of a specific page. The addition of these features could add some additional polish to the experience of using the bootstrap network, but I felt were not necessary for demonstrating the functionality of the bootstrap network.
CHAPTER IX
CONCLUSION

While further improvements could doubtlessly continue to be made for years to come, the presented implementation represents a working prototype of the COR bootstrapping network.

In review of the previously stated thesis goals, I have designed a COR bootstrapping protocol and implemented it on the AWS platform (although it could be ported to others). The ASP, relay, and client software services have all been implemented as described, and integrated to work together. I have also provided performance evaluations of the COR bootstrapping network, and suggested future directions to improve upon the current design/implementation.

Please refer to the included appendices at the end of this paper for instructions on how to install and configure the COR bootstrap network, as well as a practical demonstration of the network in action.

While the battle over privacy and anonymity rights in the online world figures to remain a hot button issue with no easy solutions, this COR bootstrap network serves to advance the cause of the rights of the individual by advancing the state of knowledge in tools available that a user can employ to protect their privacy and anonymity.
REFERENCES


APPENDIX A:
INSTALLATION AND CONFIGURATION

A.1. Library Installation

To begin, a number of libraries will be needed to be installed on the separate machines. The client machine will require the following libraries:

gcc
curl
curl-devel
openssl-devel

These can be installed using the command:

```sh
sudo yum install gcc curl curl-devel openssl-devel -y
```

To build the relays, the above libraries will need to be installed, as well as the following:

httpd24
php56
php56-mysqlnd
mysql56-server
mysql-devel
mod24_ssl
openssl-devel

These can be installed using the following command:

```sh
sudo yum install gcc curl curl-devel httpd24 php56 php56-mysqlnd mysql56-server mysql-devel mod24_ssl openssl-devel -y
```
The ASPs will need all the same libraries as the relays, except for the openssl-devel library. These can be installed using the command:

```
sudo yum install gcc curl curl-devel httpd24 php56 php56-mysqlnd mysql56-server mysql-devel mod24_ssl -y
```

### A.2. Source File Distribution

The source files to set up my implementation of the COR bootstrapping network can be found at [http://walrus.uccs.edu/~gsc/pub/master/ksnider2/src/V5.zip](http://walrus.uccs.edu/~gsc/pub/master/ksnider2/src/V5.zip). The following files will need to be distributed to their respective machines:

- A copy of client.c and the list of ASPs (asps.txt) to the client machine.
- A copy of index.html and relays.php to each ASP.
- A copy of relay.c to each relay machine.

On each ASP and relay, the following steps will need to be performed:

Move index.html and relays.php to the correct html directory, using the commands:

```
sudo mv index.html /var/www/html/index.html
sudo mv relays.php /var/www/html/relays.php
```

Change owner/group to root for index.html and relays.php, using the commands:

```
```

Ensure the correct permissions for index.html and relays.php, using the commands:

```
sudo chmod 644 /var/www/html/index.html
sudo chmod 644 /var/www/html/relays.php
```

On each relay, compile the C socket code using the command:
gcc relay.c -o relay -lcrypto -lcurl -L/usr/lib64/mysql -lmysqlclient -l/usr/include/mysql

On the client machine, compile the C socket code using the command:

gcc client.c -o client -lcrypto -lcurl

**A.3. System Configuration**

On each ASP and relay, the following steps will need to be performed:

Start the Apache web server and SQL server on each machine using the commands:

```bash
sudo service httpd start
sudo service mysqld start
```

Configure the Apache web server and SQL server on each machine to start with system boot using the commands:

```bash
sudo chkconfig httpd on
sudo chkconfig mysqld on
```

To complete the MySQL database installation and configuration on each machine, we will need to use the command:

```bash
sudo mysql_secure_installation
```

We will be presented with a number of prompts. At the first prompt, no entry is needed so just press <Enter>.

Next we will be asked to set a root password - hit 'Y', then enter your chosen password at each of the two prompts (for the rest of the example, we will use <password> to signify your chosen password).

At each of the next four following prompts, enter 'Y':

- Remove anonymous users?
Disallow root login remotely?

Remove test database and access to it?

Reload privilege tables now?

This should complete the secure installation of the MySQL service. Next we will configure the necessary database instances for the COR bootstrapping service. Enter the following sequence of commands:

```
mysql -u root -p<password> mysql
UPDATE user SET Password=PASSWORD('<password>') WHERE user='cor';
FLUSH PRIVILEGES;
create database cordb;
GRANT all privileges on cordb.* to cor@localhost identified by '<password>';
exit
```

Now that we have created the database instance we will need to create the credentials table, which is done by running the script creds.sql using the command:

```
mysql -u root -p<password> cordb < creds.sql
```

In addition, on the ASPs ONLY the script relays.sql needs to be run to create the relays table, using the command:

```
mysql -u root -p<password> cordb < relays.sql
```

Finally, on each ASP the information about the relays belonging to that ASP need to be entered into its relays table. This is done using the following commands (repeating the insert command for each relay owned by the ASP, where <address>, <location>, and <hostname> signify the IP address, location, and hostname for the respective relay):

```
mysql -u root -p<password> cordb
```
insert into relays (address, location, hostname) values ('<address>', '<location>', '<hostname>');

exit

This concludes the MySQL configuration. Next we will need to configure the appropriate self-signed certificates to allow for HTTPS connections. These steps will also need to be completed on each ASP and relay:

First navigate to the correct directory using the command:

cd /etc/pki/tls

Open the configuration file for editing using the command:

sudo vi openssl.cnf

Edit the following lines like so (substituting appropriate values for your organization, and uncommenting any lines that may be commented out):

countryName_default = US
stateOrProvinceName_default = Colorado
localityName_default = Colorado Springs
0.organizationName_default = UCCS
organizationalUnitName_default = Computer Science

Remove any preexisting certificate authority directory content using the command:

sudo rm -rf ../CA

Create a new CA using the command:

sudo misc/CA -newca

Then enter the following values at the ensuing prompts (where <password> is your chosen password, and <hostname> is the hostname assigned to the specific ASP or relay
you are currently configuring. Note that the myuccs.net domain in the email field may also differ depending on the domain assigned by your organization):

<Enter>

$password> x2
<Enter> x5
CA-$hostname$

c@-$hostname$.myuccs.net
<Enter> x2

$password>

Create a new certificate request using the command:

sudo misc/CA -newreq

Then enter the following values at the ensuing prompts:

$password> x2
<Enter> x5
$hostname$.myuccs.net
webmaster@$hostname$.myuccs.net
<Enter> x2

Sign the certificate using the command:

sudo misc/CA -sign

Then enter the following values at the ensuing prompts:

$password>

y x2
Copy and rename the new certificate to the appropriate name and location using the command:

```
sudo cp newcert.pem certs/localhost.crt
```

Generate the local private key using the command (entering `<password>` when prompted):

```
sudo openssl rsa -in newkey.pem -out private/localhost.key
```

Protect the key's permissions and remove duplicate content using the commands:

```
sudo chmod 700 private/localhost.key
sudo rm new*.pem
```

Edit the machine's recognized hosts file using the command:

```
sudo vi /etc/hosts
```

Replace the first line in this file with the line:

```
127.0.0.1 <hostname>.myuccs.net <hostname> localhost.localdomain localhost
```

Complete this process by restarting the Apache service with the command:

```
sudo service httpd restart
```

Finally, the appropriate certificates will need to be distributed to the certificate bundles on the correct machines. On each relay, copy the content of `~/etc/pki/CA/cacert.pem`. Append this content to the end of `~/etc/pki/tls/certs/ca-bundle.crt` on the ASP that owns that particular relay. Once this is done, copy the content of the same cacert.pem file on each ASP, and append it to the end of the same ca-bundle.crt file on each relay and the client machine. This concludes the HTTPS configuration of the COR bootstrapping system.
APPENDIX B:
DEMONSTRATION

The following sequence of pictures depicts the retrieval of the contents of one of the ASP pages via a 2-hop circuit, capturing the output via a special demo mode of the COR bootstrapping network:
Figure 6. Demo Sequence No. 1
Figure 7. Demo Sequence No. 2
Figure 8. Demo Sequence No. 3
Figure 11. Demo Sequence No. 6
Figure 14. Demo Sequence No. 9
Figure 16. Demo Sequence No. 11
Figure 17. Demo Sequence No. 12
Figure 18. Demo Sequence No. 13
Figure 19. Demo Sequence No. 14
Figure 25. Demo Sequence No. 20
Figure 28. Demo Sequence No. 23
Figure 29. Demo Sequence No. 24
Figure 30. Demo Sequence No. 25
Figure 31. Demo Sequence No. 26