5-2013

Tor Bridge Distribution Powered by Threshold RSA

Jordan Hunter Deyton
jdeyton@utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Jordan Hunter Deyton entitled "Tor Bridge Distribution Powered by Threshold RSA." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Jinyuan Sun, Major Professor

We have read this thesis and recommend its acceptance:

James S. Plank, Bradley T. Vander Zanden

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Tor Bridge Distribution Powered by Threshold RSA

A Thesis Presented for
The Master of Science
Degree
The University of Tennessee, Knoxville

Jordan Hunter Deyton

May 2013

Acknowledgements

I would like to thank everyone in EECS IT for a fun job while in school; my advisor Dr. Jinyuan Sun and fellow student Yue Tong for their helpful criticism; my committee members Drs. James Plank and Brad Vander Zanden; the Tor community; and, most importantly, May Yang and my parents for their support.

A little Learning is a dang'rous Thing

Drink deep, or taste not the Pierian Spring:

There shallow Draughts intoxicate the Brain,

And drinking largely sobers us again.

Fir'd at first Sight with what the Muse imparts,

In fearless Youth we tempt the Heights of Arts

While from the bounded Level of our Mind

Short Views we take, nor see the Lengths behind

But more advanc'd, behold with strange Surprize,

New, distant Scenes of endless Science rise!

Alexander Pope, *An Essay on Criticism*

Abstract

Since its inception, Tor has offered anonymity for internet users around the world. Tor now offers bridges to help users evade internet censorship, but the primary distribution schemes that provide bridges to users in need have come under attack. This thesis explores how threshold RSA can help strengthen Tor's infrastructure while also enabling more powerful bridge distribution schemes. We implement a basic threshold RSA signature system for the bridge authority and a reputation-based social network design for bridge distribution. Experimental results are obtained showing the possibility of quick responses to requests from honest users while maintaining both the secrecy and the anonymity of registered clients and bridges.

Table of Contents

List of Figures

Nomenclature

n	Number of parties involved in threshold scheme
σ_i	The partial signature of S_i
t	Threshold out of n such that $n \geq 2t - 1$
p, q	Prime numbers used in RSA
N	Public RSA modulus defined as $N = pq$
$\phi(N)$	Private RSA modulus defined as $\phi(N) = (p-1)(q-1)$
e	Public RSA key
d	Private RSA key defined as $d = e^{-1} \mod \phi(N)$
S_i	One of the bridge authority servers
d_i	The private key share of S_i

Chapter 1

Introduction

Privacy is one of the most cherished features of a free society. However, few things empower an individual more than privacy *and* anonymity. When given to members of a restricted society, privacy and anonymity are invaluable and can pave the way to greater freedom. Tor is one popular tool to help people achieve a level of privacy and anonymity over the internet, but it must constantly evolve to maintain internet anonymity for its users. This thesis seeks to augment Tor's functionality to make it reliably available for the average user.

1.1 Tor 101

Tor is a distributed overlay network that offers anonymity for TCP-based applications [1]. To the average user, it appears simply as a browser and requires little knowledge of the underlying architecture. For the purposes of this thesis, we require a more thorough understanding of how Tor works. Figure 1.1 shows how a client, Alice, would talk to a server, Bob, over the Tor network, and an explanation of the steps involved follows.

Alice downloads a version of Tor, say Vidalia, which functions as an Onion Proxy (OP). To establish a connection over Tor, Alice must first obtain an updated list of Onion Routers (ORs) from a Tor directory server (Step 1). Alice's OP then chooses at least three ORs and incrementally builds a circuit through these ORs by establishing session keys with each OR consecutively (Step 2). Once the circuit is completed, Alice can send her TCP packets

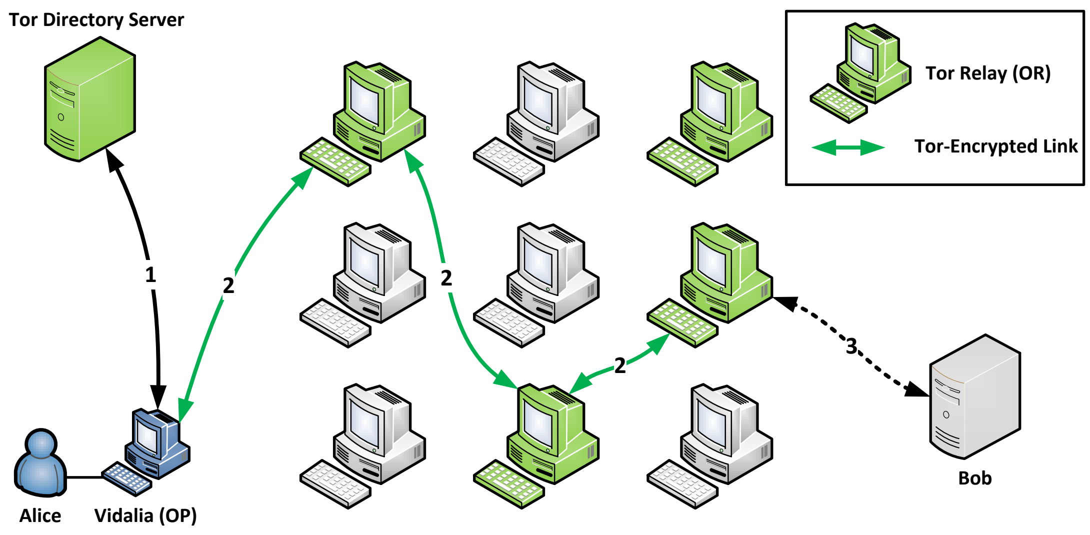

Figure 1.1: A basic example of routing in Tor.

to Bob through the circuit (Step 3). To Bob, it appears the final OR, the exit node, is establishing the communication, but Bob's responses are forwarded back through the circuit to Alice. Note that the communication between the exit node and Bob are not encrypted by Tor. The links between Alice and each OR in her circuit are encrypted in layers with session keys, but it is up to Alice and Bob to establish application-level encryption. Further details on the protocol are available in [1].

1.2 Tor Bridges

1.2.1 Background

Although the basic Tor design functions well for most people, the public nature of its infrastructure enables adversaries to easily block access to the Tor relay network. For example, consider a Tor user Alice whose ISP is controlled by the adversary. The adversary need only periodically request a list of ORs from the Tor directory servers and block all traffic destined for the IP of an OR.

To circumvent this problem, in [2], the authors introduce the concept of Tor bridge relays, or bridges. Bridges serve only as entry points into the rest of the Tor network. Unlike ORs, the Tor directory servers do not publish a list of bridges, so an adversary cannot identify all bridges with a single request. Bridges can either remain private or publish to a central bridge directory authority, which can then distribute bridges to users in a number of ways. The public bridges comprise a set of pools, each of which correspond to a different distribution strategy. As proposed in [2], the pools are as follows:

1. A time-release distribution in which there are several partitions that are only available during specific periods.
2. An IP-based distribution in which the IP of the requesting user determines which bridges he will receive.
3. A combination of the two previous schemes. Requests from a single "/24" network during the same time period will receive the same set of bridges.
4. A mailing list that is used to publish new bridges periodically.
5. An email-based distribution in which users send a request via emails from Yahoo! or Gmail. These two providers in particular require CAPTCHA input for account creation.
6. A reputation-based social network design.
7. Reserved.
8. Reserved.

In practice, there are primarily two methods of bridge distribution employed by Tor: HTTPS bridge distribution as in pool 3, and Gmail bridge distribution as in pool 5. Furthermore, as of March 2012, there were only about 1,000 public Tor bridges [3]. As of April 2013, there are just over 1,300 bridges as reported by the Tor Metrics Portal [4].

Figure 1.2 shows how bridges operate in Tor. Assuming that Alice's ISP has blocked all connections to Tor directory servers or ORs, Alice first needs to request a set of bridges from

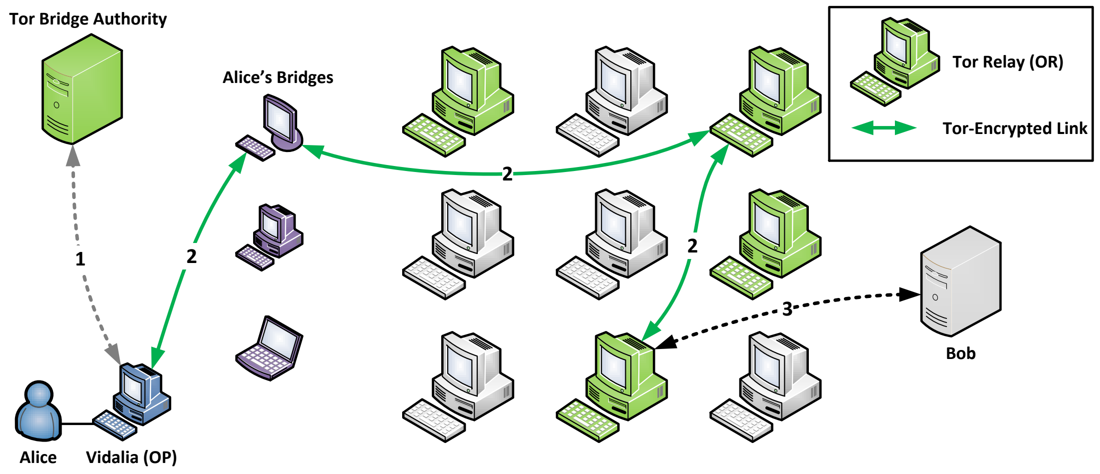

Figure 1.2: A basic example of Tor bridges.

the bridge authority (Step 1). For example, Alice can create a Gmail account and send a specifically formatted email request to "bridges@torproject.org" and will receive a response with a set of bridges. Alice will then use the IP address of one of her bridges as her entry point and will select two ORs before building a Tor circuit as usual (Step 2). Once her circuit is complete, Alice can establish a connection through Tor to Bob (Step 3).

1.2.2 Previous Work

Despite the introduction of bridges, attackers have found ways to discover and block them. For example, China successfully defeated both the HTTPS and the Gmail distribution methods in September 2009 and March 2010, respectively [5]. [6] validated this problem by launching their own bridge discovery attacks; they successfully obtained 2,365 bridges via email and HTTPS and 2,369 bridges via control of a single Tor middle router over two weeks! After a Tor user reported his private bridge relay as being blocked quickly [7], the authors of [8], built upon preliminary analysis [9] that determined that Deep Packet Inspection was to blame. Furthermore, it turned out that Tor traffic exhibits some unique features. Similar attacks have also occurred in Iran [10], but [5] suggests that tools like *obfsproxy* [11], which transforms Tor traffic into innocent-looking data, are making headway into stopping such network-level attacks.

While the fingerprint of Tor is one method used to detect bridges, the distribution strategies currently employed are similarly vulnerable. [12] shows that the need for a reputation-based social network design remains unsatisfied.

Proximax [13] is a distribution system that can be applied to Tor bridges. In *Proximax*, there are registered users who learn bridges from the system, while other users learn bridges from each other. The system determines which registered users will receive more bridge addresses to disseminate depending on their reputation. The authors also list some challenges still facing *Proximax* and other distribution schemes: censoring nations may develop cooperative agreements to share information, adversaries can imitate normal users to improve their reputation, adversaries can accrue bridge addresses to block them simultaneously in DDoS fashion, and adversaries can target the bridge authorities to learn all bridge addresses. The authors have yet to implement *Proximax* to acquire empirical results.

An alternative reputation-based scheme to *Proximax* is *rBridge*. Introduced in [14], *rBridge* is a reputation-based bridge distribution scheme that additionally offers a layer of privacy preservation for its users. *rBridge*'s underlying distribution scheme requires a set of registered users who earn credits based on the availability of their assigned bridges. Periodically, the bridge authority will distribute invitation tokens to members with high credit balances, and those users may distribute the invitation tokens to any friends by any means. However, since knowing a user's set of entry points can reduce the anonymity of that user in Tor, the creators of *rBridge* create a privacy-preserving mechanism to effectively hide the bridge selection from the bridge authority using a combination of zero-knowledge proofs and Oblivious Transfer (OT).

Besides *Proximax* and *rBridge*, to date there are no effective reputation-based bridge distribution strategies adopted by Tor. However, recall that the sixth pool of bridges maintained by Tor depends on the development of a robust, reputation-based social network scheme. We aim to contribute to filling this gap in the evolution of Tor while addressing some of the problems facing such bridge distribution schemes.

1.3 Threshold RSA

1.3.1 Background

First published in [15], RSA is a popular tool used in public-key cryptosystems. RSA can be described in a few simple steps: the key owner picks two distinct prime numbers p and q, computes $N = pq$ and $\phi(N) = (p-1)(q-1)$, picks a public key e, and computes the private key d as the multiplicative inverse of $e \mod \phi(N)$. The difficulty in compromising the private key lies in the factorization of N when it is a large number, e.g., a 2048-bit number.

Threshold cryptography as described by [16] effectively eliminates a single point of failure–the lone user with the private key–and forces a group of entities to cooperate in order to perform decryption or signatures. The classic example is one seen in movies: in order to launch a missile, two officers on opposite sides of a control panel must simultaneously insert their key into a special lock. Likewise, for threshold signatures, some minimum portion of authorized entities must cooperate to perform a cryptographic signature with their shared private key.

Following the logic of [17], a (t,n)-threshold signature scheme (or t-out-of-n) is one in which at least t out of n parties must cooperate in order to perform a valid signature for a message m. The goal of threshold RSA is to split the private key d across n parties in such a way that at least t of said parties can perform a valid RSA signature that can be verified with the public key e and the modulus N.

1.3.2 Previous Work

The first example of a (t,n)-threshold RSA scheme in literature is in [18], which provides a simple method based on polynomial interpolation to share the key in such a way that

$t-1$ of the parties cannot recover the remaining shares of the key. Extending upon this work, [19] adapted the method to be more practical[*].

[17] designed a non-interactive and an interactive protocol for verifying partial signatures. Through their scheme, parties that control shares of the key are able to verify the partial signatures of potentially compromised members and discern the bad partial signatures from the valid. [20] employed and supplemented the methods above to create a complete threshold RSA scheme. In their paper, each party's share is in turn shared among the other participants by adapting a Verifiable Secret Sharing (VSS) scheme, which allows the share to be "regenerated" when it has been compromised. Furthermore, the shares among the servers can be refreshed if the compromised party returns.

While earlier papers either ignore how to generate the RSA key itself or assume a trusted dealer would generate and distribute the n shares of d, [21] provides a distributed method to generate the shared RSA key for groups of at least three parties. At the end of their protocol, each of the n parties has a share of the secret key d, but no one knows the value of d or any of the other shares. [22] adapted that protocol to enable distributed two-party RSA key generation. Although both papers provide insight to eliminating the trusted dealer from the key generation process, the former paper introduces overhead during the process, whereas the latter paper targets only two-party RSA keys.

[*]Shamir's method [18] requires that the share space of d be a field. d is in the ring $Z_{\phi(N)}$, and polynomial interpolation is not necessarily possible. [19] generalized the scheme by requiring the share space to be a finite Abelian group. A brief sketch of their proof is available in [17], but the reader may refer to [19] for complete details.

Chapter 2

System Overview

Following the discussion of how Tor, bridges, and threshold RSA work, we now provide an overview of this thesis. The following sections explain the objectives of the thesis, the assumptions about adversarial threats, and the overall architecture of the approach.

2.1 Our Objectives

As asserted in the previous chapter, Tor bridges must adapt to the counter attacks in place. Furthermore, we must remain vigilant and prepare for future attacks on the Tor architecture. The single, overarching goal of this thesis is to help make Tor more accessible for the average internet user in censored nations. To help contribute this effort, we wish to gain insight into how to develop a robust system for Tor bridge distribution that can withstand active attacks and yet retain anonymity and privacy for the bridges and their users. Specifically, our goals are as follows:

1. The first objective is to determine if threshold cryptography is suitable and efficient for use in the Tor environment.
2. The next objective is to open the door to more advanced bridge distribution techniques that are mathematically efficient.

3. The last objective is to provide experimental results to support our arguments.

The purpose of threshold cryptography in this project is to remove a single point of failure. Previous papers([13], [14]) do not account for the compromise of the bridge authority. With the current state of their projects–and, in fact, the current state of Tor, whose lone bridge authority is in Tonga [3]–, the compromise of the bridge authority by an adversary would immediately result in the leak of all public bridge IP addresses and likely result in system downtime.

For the second objective listed, we seek to enable future research into better bridge distribution algorithms. Devoted mathematicians could perhaps derive very efficient bridge distribution schemes that would be interchangeable with the rest of our protocol, but the creation of a mathematically sound strategy is outside the scope of this thesis.

We hope that our design helps offer a consistently reliable supply of bridge IP addresses to users who need them without compromising the privacy of either the bridges or users. In other words, the bridge authorities and any eavesdroppers should not learn the IP addresses of either users or bridges from the execution of our protocol.

2.2 Threat Model

Since we are focusing on Tor and its bridges, we draw from the threat models described in [1] and [2]. In our project, we make the following important assumptions:

1. We have an initially trusted dealer controlled by the Tor Project.
2. Adversaries may take control of users, bridges, or some fraction of bridge authority servers.
3. The objective of the adversary is to either learn bridge IP addresses or otherwise render the service unusable.
4. A Tor circuit with underlying TLS offers acceptable anonymity and privacy between the two endpoints.

5. Bridges can notify the bridge authority soon, say within a day, when they have been blocked.

2.3 Threshold-based System

Recall that the sixth pool of public Tor bridges is reserved for a reputation-based social network. Since the HTTPS and Gmail distribution schemes have been defeated [5], we propose a scheme for robust bridge distribution with an underlying reputation-based social network design. Since *Proximax* [13] and *rBridge* [14] are the only two such reputation-based schemes to have been proposed in literature so far, we should first revisit them briefly to try to learn from them.

2.3.1 Challenges

Neither *Proximax* nor *rBridge* resolve the issue that the current bridge authority server [3] is a single point of failure. Because it knows all of the public bridge IP addresses, compromise would mean the adversary could block all such bridges in their own country and bring down the Tor public bridge distribution service. This is where threshold cryptography can help: increasing the number of servers the adversary would need to compromise *simultaneously* increases the difficulty of compromising the system as a whole. Furthermore, the servers could be placed in separate locations or managed by separate teams, so an adversary would have to increase their efforts dramatically to break into so many systems at once.

Another problem inherent to the bridge authority is that it knows the real IP addresses of the bridges. Ideally, this information should remain a secret shared between a bridge and its users even if the bridge authority is compromised.

Other issues include the bridge distribution strategy itself. *rBridge* [14] uses an Oblivious Transfer (OT) technique to mask a user's bridge selection. This approach forces the bridge assignment for each user to be random. As an argument against a random selection approach, suppose we have 100 bridges and hand out 3 per user. Suppose further that we have a malicious user in our set of registered users and that he has received an invitation token for behaving well. With the threat model in mind, the adversary will likely invite another malicious user into the network. If bridges are selected randomly, then these two malicious nodes are likely to learn 6 total bridges. However, a more intuitive bridge selection strategy could force the two malicious users to share some bridges. For instance, we could reduce the number of bridges known by the adversaries to 4 if they share 2 bridges. The bridge authority can enforce such a strategy if it knows who invites whom into the network.

In developing our scheme, we must also avoid incurring too much overhead. For instance, the zero knowledge proofs and OT protocols used in *rBridge* result in averages of just over 5 seconds of computation time for the user and over 17 seconds for the bridge authority during registration and bridge requests, respectively [14]. Likewise, because in *Proximax* the users disseminate the bridge IP addresses themselves [13], it could take significant time for a user to tell his friends about his new bridge IP addresses.

2.3.2 Our Contributions

First, we give the basic idea behind our approach. The concept of a reputation-based social network design is straightforward: registered users improve their reputation by not blocking bridges, and users with a good reputation can invite their friends into the network. Invitations require a valid invitation token dealt by the bridge authority, who acts as a middle man between the bridges and users. Figure 2.1 provides a brief overview of our approach.

Bob is a user that has a good enough reputation, so when he connects to the bridge authority (1), the authority responds with an invitation token (2). Bob then distributes this token to his friend, Alice (3). In order to join the network, Alice contacts the bridge authority (4). This can be done either outside Tor, e.g., via email, or using one of Bob's bridges to build a Tor circuit. Periodically, the bridges report to the bridge authority (5). When the authority has determined which bridge to assign Alice, it relays Alice's request for the bridge's IP address (6). The bridge will then respond (7) with its IP address. The

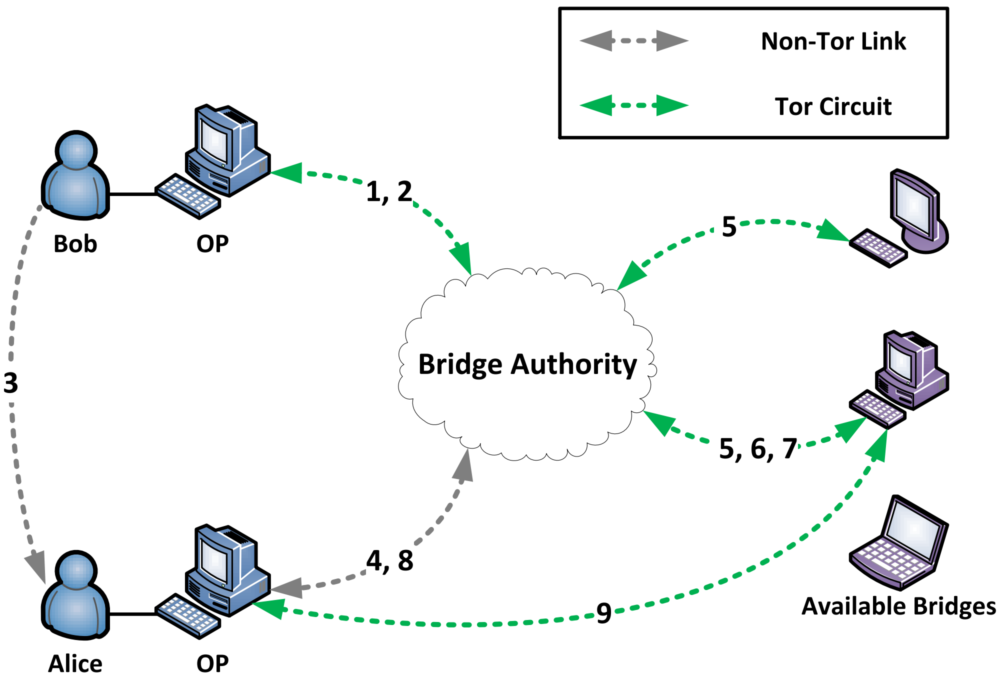

Figure 2.1: An overview of our scheme.

authority relays this message back to Alice (8). Now that Alice has the bridge's IP address, Alice can build a Tor circuit using the bridge (9).

In order to avoid keeping the bridge authority a single point of failure, we adapt the (t,n)-threshold RSA signature scheme proposed in [20]. For instance, if we have three bridge authority servers, the adversary would have to compromise two of the servers in order to recover the key of the third server and produce valid threshold signatures.

Note that the bridge authority is the middle man that the users must trust. With threshold cryptography, the bridge authority can be trusted to a greater extent as the middle man. We take advantage of this by using the bridge authority to establish trust between the client and its selected bridges. This allows the bridge's IP address to be kept secret from the bridge authority.

We also borrow the basic, non-private reputation system proposed for *rBridge* in [14] to show how a reputation-based bridge distribution scheme can operate within our protocol.

If one were to use *Proximax* as the reputation system, our protocol could be tweaked to transfer bridge IP addresses secretly to the seed users. More details on the methods we employ can be found in the next chapter.

Chapter 3

Threshold RSA-based Tor Bridge Distribution

For this thesis, we implemented a distributed, multi-threaded server that can generate a threshold RSA key-pair and can use it to authenticate itself to clients and bridges in the Tor network for anonymous bridge distribution. This chapter will describe in detail the threshold RSA scheme used, the reputation-based bridge distribution implemented, and the overarching protocol developed.

3.1 Threshold Signatures

The threshold RSA signature scheme implemented in this thesis is derived from the proactive threshold RSA protocol in [20]. This protocol supports n servers with a threshold of t where $n \geq 2t - 1$. Below, we describe each of the major components of the threshold RSA protocol in terms of our environment.

3.1.1 Key Generation

During the key generation phase, the dealer generates a basic RSA key-pair. This key will later be shared among the n servers. The steps involved in key generation are as follows:

1. Generate a 2048-bit RSA key-pair. (N, e) is the public key. $N = pq$, where $p = 2p' + 1$ and $q = 2q' + 1$ with p, q, p', and q' are prime.* The private key d satisfying $de = 1 \mod N$ will be shared among the servers.

2. Pick a global value $g = g_0^{L^2} \mod N$ where g_0 is an element of high order (a random 2048-bit number) and $L = n!$.

Although we use 2048-bit moduli for RSA key generation, this is not a restriction. In the implementation, the dealer performs all of these steps. Because the dealer knows the private key d, he can perform a valid signature later on. The dealer needs to be trusted during key generation and must forget the private key and its shares. Alternatively, the distributed protocol in [21] could remove the need for a trusted dealer. Once the base RSA key is generated, it can be shared among the participating servers.

3.1.2 Sharing

This phase splits the private key d into pieces and shares it among each of the n servers. For each private share d_i belonging to server S_i, a witness value will be computed to be used for signature verification at a later stage. Additionally, each server will be given a piece of the other servers' private shares through Verifiable Secret Sharing. The steps below are performed by the dealer for each server S_i:

1. Select a random $d_i \in [-nN^2..nN^2]$. Set $d_{public} = d - \sum_{i=1}^{n} d_i$.

2. Compute $w_i = g^{d_i} \mod N$.

3. Construct the polynomial $f(x) = a_t x^t + ... + a_1 x + d_i L$ with random $a_t, ..., a_1 \in [-nL^2N^3..nL^2N^3]$.

4. Compute $f(j)$ for $1 \leq j \leq n$.

5. Compute $b_t = g^{a_t} \mod N, ..., b_1 = g^{a_1} \mod N, b_0 = g^{d_i L} \mod N$.

*p and q are considered strong primes. This is required by the threshold RSA protocols in [20] and [17].

6. Send to server S_i his secret share d_i.
7. Broadcast the witness w_i and $b_t, ...b_0$.
8. Send to server S_j the value $f(j)$.

Once these data have been distributed accordingly, each server must verify the information he receives. The steps below are performed at each server S_i:

1. Verify that $g^{f(i)} = \prod_{j=0}^{t} (b_j)^{i^j}$. Request the dealer to publish $f(i)$.
2. The dealer will broadcast all requested $f(j)$s.
3. For each published $f(j)$, perform Step 1. If any verifications fail, the dealer is disqualified.

If the verification of the $f(i)$s fail, the dealer is disqualified, and the process needs to restart. Once this phase has been completed successfully, the dealer is no longer a part of the system, and the servers can begin signing messages. The dealer's knowledge of this threshold RSA key should be erased immediately.

3.1.3 Signature Generation

In a typical shared RSA signature scheme, a valid signature can be generated by multiplying the individual partial signatures on a message m. The same holds for threshold RSA signatures as defined in [20]. Thus, to perform a complete signature on a message m, we must perform the following steps:

1. Each server S_i publishes its partial signature $\sigma_i = m^{d_i} \mod N$.
2. The complete signature on m is $m^{d_{public}} \prod_{i=1}^{n} \sigma_i \mod N$.

Verifying the signature is the same as with normal RSA signature verification. If c is the ciphertext generated by the signature operation on a message m, then the verifier computes $c^e \mod N = m^{de} \mod N = m$ with the public key (N, e).

3.1.4 Verifying Partial Signatures

The benefit of this protocol is that an individual partial signature with share d_i can be validated using the witness w_i generated in the previous section on sharing. If the partial signature on m is invalid, then the share d_i can be recovered by at least t of the n servers. The interactive verification protocol discussed in [17] and listed below can be used to verify a partial signature. Suppose server S_i wishes to verify the partial signature $\sigma_j = m^{d_j} \mod N$ of server S_j. The steps below detail this process:

1. S_i chooses random $x, y \in [0..N]$ and computes $Q = m^x w_j^y \mod N$. S_i sends Q to S_j.
2. S_j computes $A = Q^{d_j} \mod N$ and sends A to S_i.
3. S_i verifies that $A = \sigma_j^x w_j^y \mod N$. If not, then σ_j is an invalid partial signature on m.

The last step is possible because of the arithmetic properties of Z_N. In other words, $\sigma_j^x = (m_j^d)^x = (m^x)^{d_j}$. This verification protocol is flawed in that S_i could send a different message m' to S_j, who would then effectively reveal his signature σ_j' on m' in step 2. The authors in [17] remedy this with the revised protocol below:

1. S_i chooses random $x, y \in [0..N]$ and computes $Q = m^x w_j^y \mod N$. S_i sends Q to S_j.
2. S_j computes $A = Q^{d_j} \mod N$ and sends a commitment to A to S_i.
3. S_i sends the values x and y to S_j.
4. S_j verifies the value of Q with its knowledge of x and y. If Q is valid, S_j sends A to S_i.

5. S_i verifies the commitment of A and that $A = \sigma_j^x w_j^y \mod N$. If both are valid, the partial signature σ_j is valid.

3.1.5 Share Reconstruction

If S_js partial signature σ_j on a message m is incorrect, then it is possible that S_j has been compromised. In a typical n-out-of-n RSA scheme, a single compromised server would render the entire system useless. However, due to Verifiable Secret Sharing (see steps 3-5 of the protocol in the above section Sharing), S_js private share d_j can be recovered by t or more of the other $n-1$ servers. The reconstruction process for a single share d_j is detailed here:

1. Each uncompromised server S_k broadcasts $f_j(k)$.
2. Server S_i finds a set I of $t+1$ indices where $\forall x \in I$, $g^{f_j(x)} = \prod_{y=0}^{t}(b_y)^{x^y}$.
3. S_i chooses a prime $P > 2nN^2$ and computes $d_j = \sum_{x \in I} f_j(x) \prod_{y \in I, y \neq x} \frac{-y}{x-y} / L \mod P$.

3.1.6 Share Refreshing

Once the share reconstruction protocol has run to completion, the key share of the compromised server will be known to the other servers. When the compromised server is restored or replaced, it cannot simply reclaim its key share. The naïve approach would be to regenerate the keys. However, this has disadvantages, including expiring signatures, server downtime, and having to re-publish the public key. This is particularly unfortunate with respect to Tor bridge distribution because new users or those who are in the process of acquiring bridges may have difficulty acquiring the new public keys.

To avoid having to regenerate the shared RSA key, the author of [20] provides a share refreshing protocol. This enables the servers to re-distribute their key shares among themselves. Also, the refreshing protocol repairs the VSS backups to reflect the changes in the key shares. Their protocol as detailed below is performed for each server's share of the private RSA key d:

1. Server S_i splits his private share into n pieces:
   (a) Server S_i randomly picks $d_{i,j} \in [-N^2..N^2]$ for $1 \le j \le n$.
   (b) S_i sets $d_{i,public} = d_i - \sum_{j=1}^{n} d_{i,j}$.
   (c) S_i sets $g_{i,j} = g^{d_{i,j}} \mod N$ for each j.
   (d) S_i sends $d_{i,j}$ to S_j.
   (e) S_i publishes $d_{i,public}$ and each $g_{i,j}$.

2. Another server S_j verifies S_is published data:
   (a) S_j verifies that his new private share $d_{i,j} \in [-N^2..N^2]$.
   (b) S_j verifies that $g_{i,j} = g^{d_{i,j}} \mod N$.
   (c) If either test fails, S_i must make $d_{i,j}$ public and set $g_{i,j} = g^{d_{i,j}} \mod N$.
   (d) If S_i fails to make these values public, his share d_i is reconstructed.

3. S_j verifies that S_i's old and new shares are the same:
   (a) S_j verifies that $w_i = g_{i,public} \prod_{j=1}^{n} g_{i,j} \mod N$.
   (b) If the test fails, S_i's share d_i is reconstructed.

4. S_i gets his new share:
   (a) S_i computes $d_i^{new} = \sum_{j=1}^{n} d_{j,i}$.
   (b) $g^{d_i^{new}} \mod N = \prod_{j=1}^{n} g^{d_{i,j}} \mod N$ is already public.

(c) S_i shares d_i^{new} via the VSS sharing phase (see steps 3-5 of the protocol in the above section Sharing).

(d) The VSS results in a public value $b_0 = g^{sL} \mod N$ where $s = d_i^{new}$.

5. The other servers verify the VSS of S_i's private key share d_i^{new}:

    (a) If S_i does not perform VSS with his secret, or if $(g^{d_i^{new}})^L \neq g^{sL} \mod N$, each other server S_j publishes $d_{j,i}$.

    (b) If some server S_j does not expose $d_{j,i}$, he is compromised, and his d_j is reconstructed.

3.2 Bridge Distribution

Now that we have described in detail the threshold RSA scheme in terms of bridge authority servers, we will now describe the bridge distribution scheme implemented in our program. As mentioned before, we base our bridge distribution on *rBridge* [14]. *rBridge* can be split up into two major parts. First, you have the bridge distribution method, and then there is the privacy preserving mechanism. Unfortunately, the privacy preservation in *rBridge* comes at the cost of increased computation time. Furthermore, due to the oblivious transfer protocols used, each user gets a random set of bridges from the bridge authority. A perfect bridge distribution system would offer the privacy of *rBridge* but give the bridge authority more control over who gets which bridge in a timely fashion. Thus, we only borrow *rBridge*'s basic bridge distribution protocol as a starting point for more advanced protocols.

3.2.1 Bridges

Recall that a Tor bridge is a user's gateway through or a detour around censorship devices. Volunteers who wish to offer their machine's services as a bridge register their computers with the bridge authority. Once they have done so, the bridge authority typically learns their IP address. When a bridge is running, users that the bridge authority points to the bridge can use the bridge to establish typical Tor circuits to the rest of the world. Thus, the substance of the bridge distribution system is the set of bridges. Bridges are not necessarily as fast as Tor routers, so the number of users that can connect to any bridge is limited both by the availability of the bridge and the amount of bandwidth it offers. For our project, as in *rBridge*, we only hand out a bridge's IP address to a select few users.

3.2.2 Clients

When a client registers with the bridge authority, he receives a random selection of a fixed number of bridge IP addresses. Once the client has bridges, he can try connecting to them to establish Tor circuits to the outside world. Throughout its lifetime, the client will connect to bridges and seek to replace any bridges that his local censor blocks. Additionally, a client can invite his friends to join the network. How the client acquires bridges and invitations to send to his friends depends on his reputation.

3.2.3 Credits

Clients earn credits based on how long the bridges that they know remain unblocked. For each bridge a client acquires, there is a window from T_0 to T_1 during which he can earn credits at a given rate of ρ. For instance, if a client receives a bridge on March 12, his window for earning credits from that bridge might be from May 12 to December 12. If the bridge is blocked before May 12, the client will receive no credits for the bridge. If a client has earned enough credits by having unblocked bridges, then they can afford to purchase a new bridge should one of their current bridges be blocked.

3.2.4 Invitations

If a client has enough credits from having unblocked bridges, then he may receive an invitation token from the bridge authority servers. Periodically, the bridge authority will select a handful of tokens to hand out to clients with high enough reputations, or high credit balances. Of course, the number of unblocked bridges available limits the number of clients that can exist at any given time. Also, to prevent a malicious client from hogging the invitations and sending them to other malicious nodes or Sybils, the bridge authority will randomly select who gets an invitation token.

3.3 Our Implementation

Before we delve into specific protocol details, we provide the basic idea inspiring this protocol. Tor, both in its current form and as for both *Proximax* and *rBridge* has only a single bridge authority server that knows the IP addresses of each registered bridge. This makes the entire set of non-private bridges vulnerable: if an adversary compromises the bridge authority, they can discover every bridge IP address available at that time, and the adversary could bring down the infrastructure. An ideal bridge authority would not only be resistant to compromise, but if compromised, it could still keep the bridge IP addresses secret and continue to provide some level of service to the clients.

One sure way to solve the latter problem is to have more than one bridge authority. By using a (t,n)-threshold RSA signature scheme, the bridge authority servers can still operate as long as t servers remain uncompromised. This gives those supporting the Tor infrastructure time to determine and resolve the problems with the compromised server.

The way we keep bridge IP addresses secret is by using the trusted t-out-of-n bridge authority servers to relay messages between bridges and clients, who should all connect to the authority via Tor circuits. Although the bridges and the clients do not have prior knowledge of each other, the bridge authority can act as a trusted middle-man during a typical Diffie-Hellman (DH) key exchange. Figure 3.1 shows a simple example of how Diffie-Hellman key exchange works with our protocol.

Alice sends her signed public DH key (message m) to the server. The servers check Alice's signature, and, if it passes, they create a full signature on the message to Bob. Bob will then validate the full signature on the message sent to him by the bridge authority servers. Once satisfied, Bob can create his own DH key and encrypt a secret message s with the symmetric DH key. Bob sends back a signed message m' containing his public DH key and the encrypted secret message. Again, the servers will validate Bob's signature, sign the message with each of their partial keys, and then relay the fully-signed message m' to Alice. Alice can then validate the full server signature and decrypt the secret message from Bob.

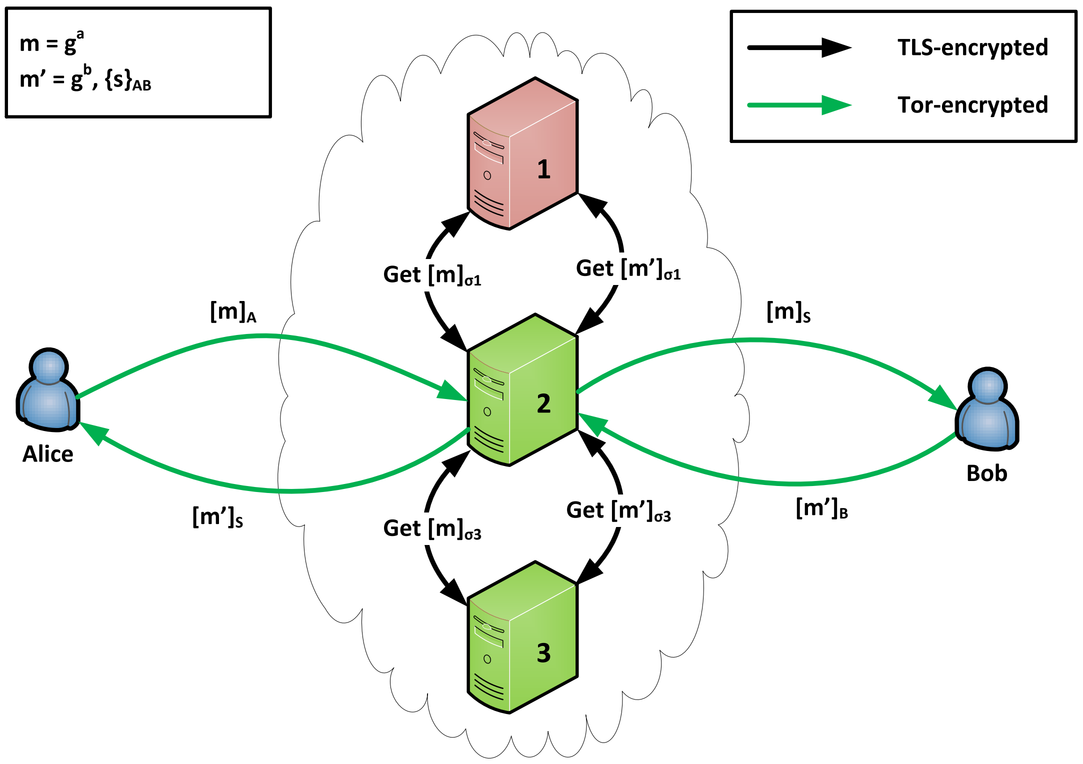

Figure 3.1: Diffie-Hellman key exchange with a trusted server infrastructure.

In a typical Diffie-Hellman key exchange, the parties need to watch out for the middle man. This is still the case, as someone in the right place between Alice (or Bob) and the servers can impersonate Alice and perform the usual MITM attack. With our protocol, Alice and Bob will both have RSA key-pairs to authenticate themselves to the servers, preventing the MITM attack. Replay attacks where an adversary replays a previous message by Alice can be prevented by typical countermeasures like session tokens, timestamps, and nonces, but for the sake of simplicity, we omit them from our protocol.

Keeping this model for DH key exchange in mind, the sections below describe in greater detail the different events and messages within our protocol.

3.3.1 Bridge Registration

Before any clients join the network, a bridge distribution system needs to have a set of bridges. Bridges can register with the bridge authority by sending a signed message to one of the bridge authority servers. Figure 3.2 shows the process of bridge registration

If a user Bob would like to offer his services as a bridge, he must first generate an RSA signature key. Then, he needs to establish a Tor circuit and connect to one of the bridge authority servers, say S_i, and send a signed copy of his public key. The servers should validate his public key before creating a unique ID_B for Bob. When server S_j validates Bob's public key, it will send its partial signature $[ID_B]_{\sigma_j}$ to S_i. Once S_i has gathered all of the partial signatures, he can generate the full signature $[ID_B]_S$ by computing the product of all partial signatures, including the partial signature with the public key share d_{public}. Once Bob receives the reply from S_i, he can validate the signature using the public component of the threshold RSA key and store his pseudonym ID_B.

3.3.2 Bridge Reporting

In future connections to the bridge authority, Bob will prefix messages with ID_B. This allows the bridge authority to identify his otherwise anonymous connection. However, because Bob is connecting to the bridge authority through Tor, the bridge authority does not know his

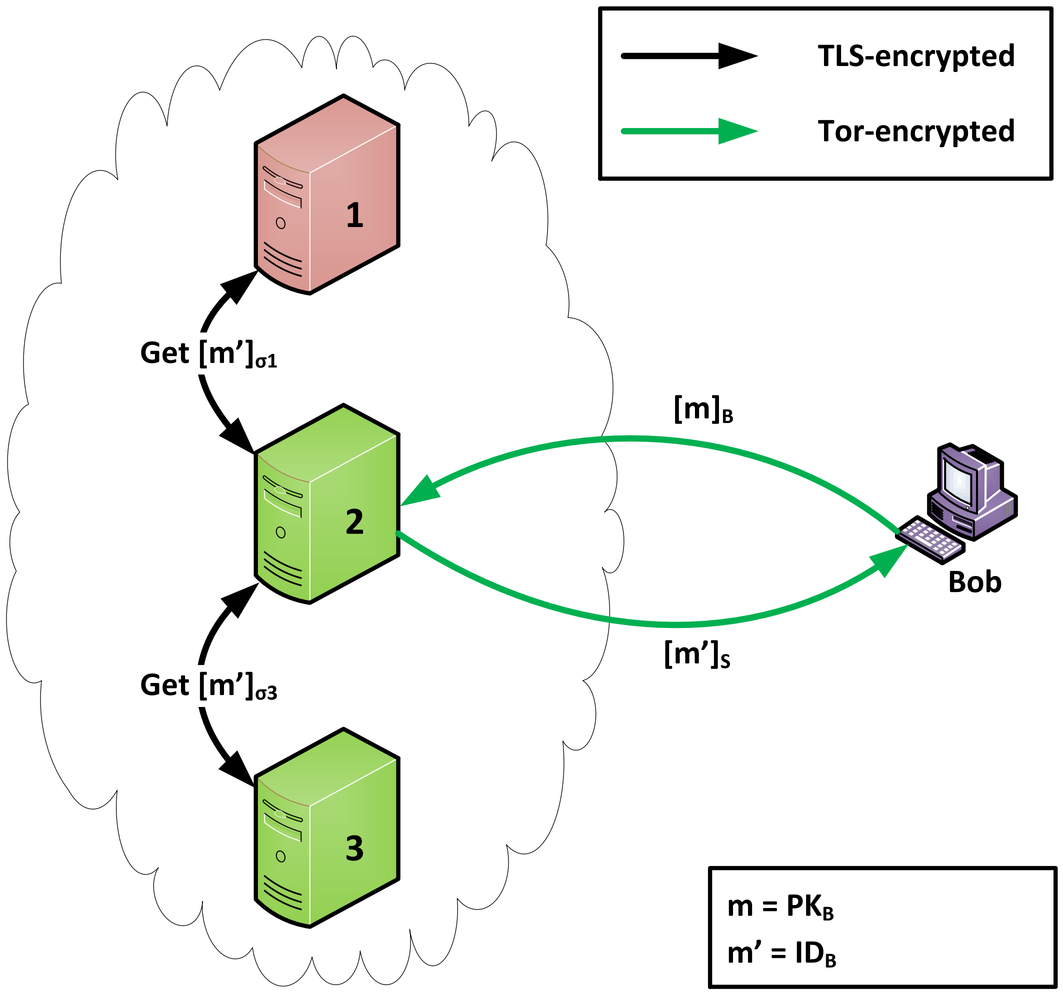

Figure 3.2: The bridge join protocol.

IP address. Hence, reports from the bridges are critical part of our protocol. Periodically, a bridge should report to one of the bridge authority servers. Figure 3.3 gives the messages involved in a bridge report.

When reporting to the bridge authority, Bob merely sends a signed copy of his identity ID_B. If the servers have any requests waiting for Bob, then the server connected with Bob will send them. The important part of each request is the public component g^a of the client's DH key. However, if the server only sent $[g^a]_S$ to Bob, a malicious server could replay the

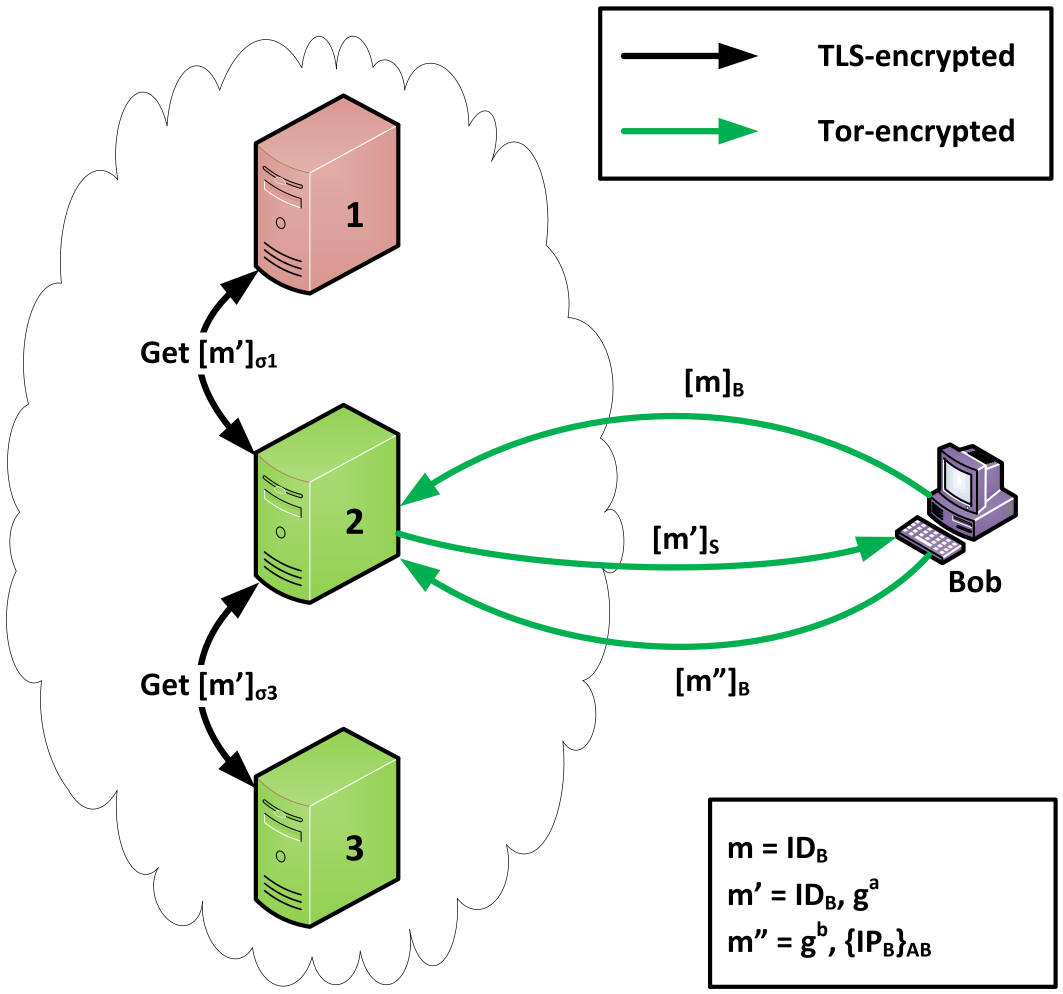

Figure 3.3: The bridge report protocol.

same fully-signed message to any bridge that connects to it. To protect Bob's IP address from being discovered by replaying old requests, each request $m' = [ID_B, g^a]$ must contain ID_B before Bob will accept its authenticity.

After verifying the IP request from the bridge authority, Bob can create a DH key with a random secret value b and the public g^a. Bob will encrypt his IP address with this secret key shared with the client and reply to the authority with a signed message $m'' = g^b, \{IP_B\}_{AB}$. The servers will then do further processing of Bob's response, but Bob is no longer needed.

Bob's connection will eventually time out, and he will report back to a different server at a later time.

3.3.3 Client Invitations

Once we have a set of bridges available to offer to clients, we can invite clients. At first, the system needs to distribute tokens to seed clients. The seeds may be known honest users, NGOs, or communities of users. For example, in *rBridge* [14], the authors consider using Chinese users on Twitter as seeds, because their use of Twitter implies an interest in censored internet resources.

Other than being selected to start the growth of the network, seeds operate as normal clients would. When they accrue enough credits, they may be given invitation tokens. The bridge authority servers periodically generate and agree on tokens, which are then doled out to randomly-selected clients with credits above a certain amount. In our protocol, these tokens are sent to selected, high-reputation clients when they report to the bridge authority as in Figure 3.5.

3.3.4 Client Registration

When a Client receives an invitation token from a friend or acquaintance who is already in the network, they can redeem the token to register with the bridge authority and receive a set of bridge IP addresses. Figure 3.4 shows the messages used during the registration process.

Before a client like Alice can register with the bridge authority, they must first obtain a token from a registered user. For a token to be redeemed, Alice needs some way to prove that the original token owner intentionally sent it to her. The way our protocol does this is to have the token owner, say Trent, use his RSA key to sign the message $[token, PK_A]$, where PK_A is Alice's public key. When Alice sends her registration request to server S_i, she will need to provide this signed message from Trent proving that Trent gave her the invitation token. In order to establish a secret communique with any bridges, Alice also must sign and

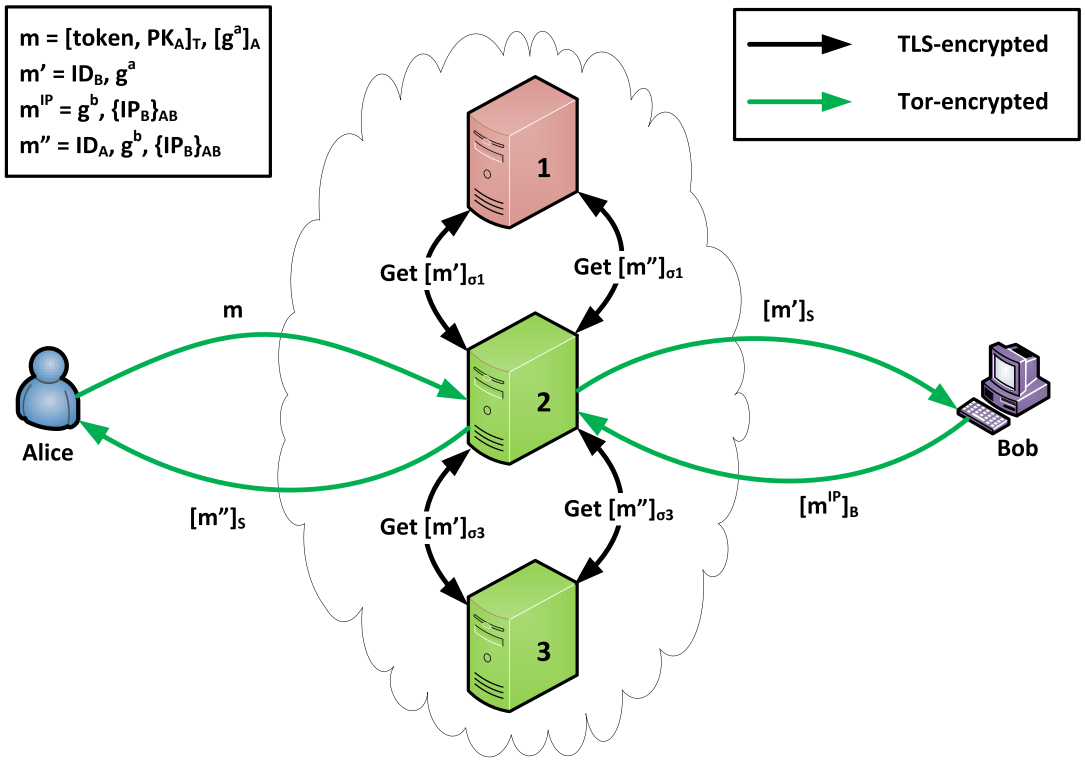

Figure 3.4: The client registration protocol.

send the public component of her DH key, g^a. Thus, the first message from Alice to S_i is $m = [token, PK_A]_T, [g^a]_A$.

The first thing S_i must do is verify the signatures on $[token, PK_A]$ and $[g^a]$. If the signatures are authentic, S_i will create ID_A for Alice and select a set of bridges. S_i will then relay the message from Alice, ID_A, and a list of messages designated for the selected bridges to the other servers. For example, S_i will select the bridge with ID_B. The message it will send to each S_j will be $[token, PK_A]_T, [g^a]_A, [ID_B, g^a]_{\sigma_i}$.

As each of the servers validate the messages and agree on the bridge selection, they will broadcast their partial signatures on each message designated for a bridge, e.g., $m' = [ID_B, g^a]_{\sigma_j}$. When all partial signatures for a particular bridge request have been published, the next server that Bob contacts can relay Alice's request for an IP address to Bob. Once all encrypted bridge IP addresses have been acquired and validated, the servers will send their partial signatures on $m'' = ID_A, g^b, \{IP_B\}_{AB}$. When S_i, receives all of the partial signatures, it can compiled the full signature on m'' and send it to Alice.

Once Alice receives $[m'']_S$, she can validate the bridge authority's signature, store ID_A, and decrypt each of the encrypted bridge IP addresses using her private DH key component a and each g^b.

3.3.5 Client Reports and Requests

The process of reporting to the bridge authority and requesting a replacement bridge is very similar to the registration process. Periodically, clients are expected to report to the bridge authority. Once connected, they will receive an update on their credit balance along with any invitation tokens selected for them. Provided the client has enough credits to purchase a new bridge, the bridge authority will select a new bridge and request an IP address from the bridge. Figure 3.5 shows this process in detail.

Alice will contact a server S_i with the signed message $[m]_A = [ID_A, g^a]_A$. If Alice has any tokens waiting for her, S_i will relay Alice's message to the other servers and ask for partial signatures on her list of tokens, m'. S_i will send to Alice a full signed list of her tokens.

If Alice needs new bridges and has earned enough credits, S_i will select some bridges for Alice and broadcast the associated bridge IP requests like $m'' = ID_B, g^a$ along with its partial signature. Again, the next server that is connected to selected bridge Bob will relay the fully-signed bridge IP request to Bob. When Bob responds, the server will broadcast Bob's reply. Once all bridge responses are accounted for, the other servers send S_i their partial signatures on $m''' = ID_A, g^b, \{IP_B\}_{AB}$. S_i then compiles the full signature on m''' and responds to Alice. Alice will then be able to decrypt the selected bridge IP addresses.

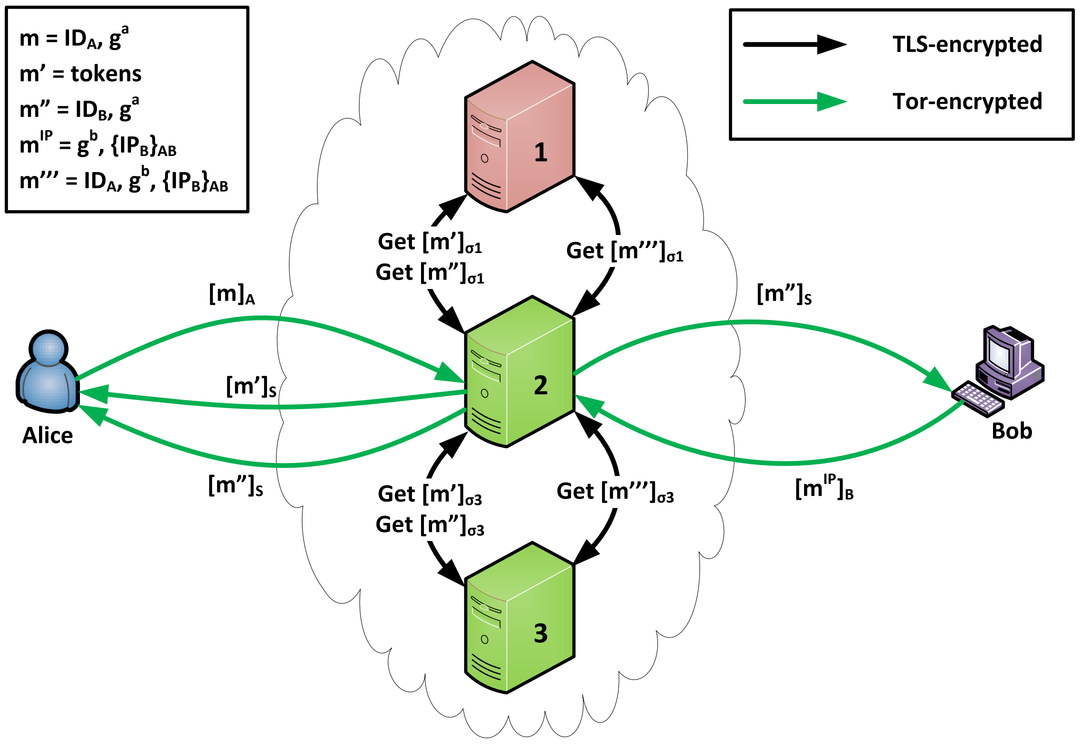

Figure 3.5: Client reports, requests, and invitations.

Chapter 4

Results

We implemented our protocol for three servers, yielding a 2-out-of-3 threshold RSA signature system, and simulated an environment with hundreds of clients and bridges. This chapter discusses specific implementation and experimental details along with the experimental results we obtained.

4.1 Implementation Details

Because BridgeDB, the current bridge distribution database backend [23], is written in Python, we implemented our protocol using Python 2.7.3. This version of Python supports Unix style sockets and SSL up to TLSv1 based on OpenSSL [24]. For communications between the servers and the dealer, we use sockets wrapped with TLSv1 and 2048-bit RSA keys and certificates. Thus, it is expected for the servers (and the dealer during key generation) to have confirmed the authenticity of each other's certificates.

All RSA keys are 2048-bit keys. This is not a restriction, and key sizes can be changed easily to reflect changes in security practices. To generate and use RSA keys either on the dealer or on bridges and clients, we use PyCrypto 2.6 [25]. When signing a message, we first hash it with SHA-256, perform the signing operation on the message hash, and then we append the signature to the end of the message. For non-threshold RSA signatures, we use PKCS#1 RSASSA-PSS (support provided by PyCrypto). For threshold signatures, we perform the standard RSA operations with the private key shares using gmpy2's interface [26] to the The GNU Multiple Precision Arithmetic Library (GMP) [27].

For the Diffie-Hellman key exchange, we follow the examples in the Tor specification [28]. Specifically, we use key lengths of 320-bits, with the public components $g = 2$ and p set to the 1024-bit safe prime in Section "6.2 Second Oakley Group" of RFC2409 [29]. To encrypt the IP address, we use 128-bit AES in ECB mode. ECB should not be used to encrypt multiple blocks (repeated plaintext blocks will have the same ciphertext), but an IP address (and a port) can fit easily into a single 16-byte block.

4.2 Experiment Details

Because we implemented *rBridge*'s basic reputation scheme, we wanted to test their reputation scheme both before and after a change. We also adapted their selected parameters, with a few changes. Table 4.1 shows the major parameters we used.

Instead of growing the bridge base as is done in [14], we keep a static set of 200 bridges. This is a more realistic scenario, since the authors of Tor would likely not put all their bridges into one distribution strategy. We are also interested in how the user base grows, so we only use 10 honest seeds rather than trying to overwhelm the set of bridges at the beginning. The last change from *rBridge* is that we move the credit earning window closer to the initial assignment. We do this because we do not simulate more advanced attacks like the event-based attacks in [14] but rather focus on "dumb" attacks (blocking bridges immediately).

Table 4.1: Selected parameters.

Number of seeds	10	Credits per day	1
Initial bridges	200	T_0	5 days
Chance for malicious client	0.05	T_1	305 days
Bridges per client	3	Credits per bridge	45
Clients per bridge	40	Invite threshold	236 credits
Days to simulate	720	Invitation period	7 days

We simulate two years of running time for the system at a rate of 10 seconds per day and gather results from the servers. We simulate two years because the authors of [14] selected the credit threshold for invites at 236. If an honest user has all 3 unblocked bridges, the first time he could afford an invitation would be nearly 80 days. We also had to throttle back the number of invitations a single user could be given at any time to prevent the system from being bombarded with hundreds of join requests at the same time.

4.3 Results

Our results were obtained by running the scripts on a LAN. We used the VNC machines vnc0.eecs.utk.edu through vnc16.eecs.utk.edu. The first three machines, vnc0 to vnc2, were used as servers, vnc3 to vnc9 hosted bridges, and vnc10 to vnc15 ran clients. For experiments involving the dealer, we ran the dealer on vnc3. Each bridge hosted clients on a specific port in the range 40000 to 50000. Each machine has an Intel Xeon X5550 2.67 GHz processor with 12GB 1066MHz PC3-8500 RAM, and runs Ubuntu 12.04 (Precise Pangolin) x64.

4.3.1 Key Generation

Before we tested the performance of the threshold RSA scheme in practice, we simulated the speed of the key generation phase. This includes all communications between clients and servers. We ran one test where the dealer always broadcasts an invalid $f(j)$ (step 8 in the first protocol of Section 3.1.2). The values in table 4.2 were averaged over 100 trials, with a new RSA key generated in each trial.

The generation time shows how long it takes for the dealer to generate all of the key data, including the witnesses and VSS shares. At 0.340 seconds on average, this is the longest section of the key generation process.

Transmission time records how long it takes to send all of the key data to each server and receive a response. Responses will be either “OK” or a request to publish $f(j)$ values from

Table 4.2: Performance of bridge distribution protocol.

Average Times (seconds)	
Generation	0.340
Transmission	0.131
Validation	0.168
Average Traffic (bytes)	
Transmission	44180
Validation	49893

the VSS protocol. In other words, it includes the time it takes for the servers to validate the key data they receive.

The validation period includes the time it takes to transmit the requested data to each server, the time for each server to validate the data, and for them to respond. Because we modified the $f(j)$s going to each server, this validation protocol produced the maximum amount of data to be sent and processed.

The next table shows the amount of data sent from the dealer throughout the experiment. This shows that at best each server need only process about 14.4 KB of data, while at worst a server must process about 30.6 KB of data.

From this experiment, the key generation process is clearly very fast. With threads handling the communications from the dealer and each server working independently, the process is nearly instant.

4.3.2 Bridge Authority

We then stored a functioning threshold key and its shares for testing the speed of the protocol when all servers are functioning properly. For convenience, when started, each server would read its key share data from a file to avoid regenerating the key each time. We ran two experiments to evaluate the performance of the protocol. The first experiment uses the *rBridge* reputation system along with its random selection of bridges. In other words, when a user joins the network, he receives 3 randomly-selected bridges.

The second experiment changes the selection of bridges slightly: when a client invites another client, the new client will receive all but one of its inviter's bridges. The remaining bridges selected up to the limit of bridges per client are randomly selected. The idea is to keep the bridge selections for a group of friends from affecting unrelated clients. This is not necessarily a perfect solution, but the idea is to show that bridge distribution algorithms are interchangeable with our protocol.

Table 4.3 shows the performance of the bridge distribution protocol for both experiments. All measurements were made at the server. Our protocol dictates that the first server to be contacted is the same server that responds, so we took these measurements at each server.

The bridge join figures measure the first instant a message is received from a bridge to the moment the server sends out the fully signed message welcoming the bridge into the network. There is only one brief exchange between the bridge and its contact within the bridge authority, thus it is the fastest of these protocols.

Client report measures the amount of time it takes for the client to get a response when reporting to the servers. This includes the servers' getting a full signature on the tokens that should be sent to the client. Although the tokens have already been selected, the full signature is not computed until the client connects in case more tokens are selected for the client. This portion of the protocol is very fast, showing again how minimal the computation time of threshold RSA signatures can be.

Table 4.3: Performance of bridge distribution protocol.

	Method Averages	
Event	Random	Inherited Random
Bridge Join	0.112	0.139
Client Join	3.219	2.707
Client Report	0.232	0.227
Client Request	3.663	3.299
Non-malicious Clients	963	752

Client join represents the amount of time it takes from the first contact from the client to the sending of the final, fully-signed message containing the client's ID and his set of bridges. The numbers here are elevated because it is possible that one or more of the bridges selected for a client are not connected to the server. Thus, the client will have to wait a little while. This can be a problem if bridges do not report as frequently as they do in our experiment (every 3 seconds). We could select bridges from those that are at that moment reporting to the server, and that would increase the average response time. However, because the client is not a part of the network yet, he *must* wait on the servers to select and query all three bridges. Thus, this is one of the slower parts of our protocol.

The last event we discuss are the client requests. Requests are considered part of the reports, but they do not always occur. For instance, sometimes a client needs to report but does not require a new bridge. These figures measure the amount of time from the first response during the report to sending the selected, encrypted bridge IP address(es) to the client. Note that the request times are longer than the recorded client join times. This could have multiple causes. First, as bridges are blocked, their IP addresses become more scarce, and with more clients making requests at the same time, the 10 threads processing requests at each server might become overwhelmed. Another issue is that, as more malicious clients register and block bridges, any bridges they block that are selected for registering or requesting clients will cause a new bridge selection.

It is interesting to note that the second bridge distribution method (inherited random) seems slightly faster with client joins and client requests. When running the experiments, it seemed that the latter distribution entered a state of equilibrium and quit growing. The random distribution had 66 out of 200 unblocked bridges at the end of the experiment, while the inherited random distribution had 63 out of 199 bridges. Both sets had almost the same number of malicious clients by the end (45 and 46, respectively). Although performance did not improve much, the appearance of graphs showing all of the clients and their parents in the network do vary greatly. With the latter method, trees are much less uniform in size, and groups of blocked clients (honest clients with no bridges) seem to develop nearer each other. The reader is referred to the appendix for the graphs.

Chapter 5

Conclusion and Future Work

We successfully implemented a threshold RSA signature system that distributes Tor bridges to clients via a reputation-based distribution scheme. We gathered results showing that threshold RSA signing is computationally fast, although it may introduce communication overhead. It also removes the single point of failure for the Tor bridge infrastructure and allows us to implement a privacy-preserving protocol for handing bridge IP addresses secretly to users in need.

Problems with this approach include the added communication overhead. More research and more practical experiments would need to be performed to ensure that threshold RSA, or rather redundant servers that operate as a single unit, is feasible in practice. This includes a more realistic simulation of a global environment, meaning possibly significant delays and packet loss between the servers.

Another problem is that bridges may report infrequently to the bridge authority, thus slowing down our protocol. There are a number of ways this might be addressed, including selecting bridges that are already connected and reporting to one of the servers and, during the client report stage, have clients send, in advance, a Diffie-Hellman key for their next request.

A more technical problem is the process of secure padding with threshold RSA signatures. In practice, RSA schemes require messages to be padded before being manipulated by the private or public RSA key components. For signatures, this includes schemes like RSASSA-PSS (Probabilistic Signature Scheme). Additional research by more cryptographically-minded individuals may be able to show such padding schemes to be either compatible or incompatible with threshold RSA.

One particularly useful advantage of our scheme is that more advanced bridge distribution schemes can be incorporated to better distribute bridges among clients with minimal impact on honest clients when malicious clients infiltrate the network. Because bridges and clients are associated with identifiers and not IP addresses, better algorithms can use the known relationships between clients to distribute bridges in a mathematically efficient way without compromising the bridge and client's IP addresses. Future research could develop schemes that can outperform either of the bridge distribution algorithms implemented for this thesis.

We hope that this approach opens the door to better, more reliable, and more secure Tor bridge distribution schemes. Such a threshold-based scheme allows much more flexibility for developing various bridge distribution algorithms without making the bridges themselves vulnerable, thus making Tor more accessible for the otherwise restricted, honest users.

Bibliography

[1] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," DTIC Document, Tech. Rep., 2004. 1, 2, 9

[2] R. Dingledine and N. Mathewson, "Design of a blocking-resistant anonymity system," 2007. 3, 9

[3] K. Loesing, "What if the tor network had 50,000 bridges?" The Tor Project, Tech. Rep. 2012-03-001, March 2012. [Online]. Available: https://research.torproject.org/techreports/bridge-scaling-2012-03-09.pdf 3, 9, 10

[4] Tor Project. (n.d.) Tor metrics portal. [Online]. Available: https://metrics.torproject.org 3

[5] R. Dingledine, "Ten ways to discover tor bridges," Technical Report 2011-10-002, The Tor Project, October 2011. https://research. torproject. org/techreports/tenways-discover-tor-bridges-2 11-1-31. pdf, Tech. Rep., 2011. 4, 10

[6] Z. Ling, J. Luo, W. Yu, M. Yang, and X. Fu, "Extensive analysis and large-scale empirical evaluation of tor bridge discovery," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 2381–2389. 4

[7] Tor Project. (2011, Oct.) Ticket #4185: Bridge easily detected by gfw. [Online]. Available: https://trac.torproject.org/projects/tor/ticket/4185 4

[8] P. Winter and S. Lindskog, "How the great firewall of china is blocking tor," *Free and Open Communications on the Internet, Bellevue, WA, USA*, 2012. 4

[9] T. Wilde. (2012, Jan.) Tor bug 4185 testing and report. [Online]. Available: https://gist.github.com/twilde/da3c7a9af01d74cd7de7 4

[10] R. Dingledine. (2011, Sep.) Iran blocks tor; tor releases same-day fix. [Online]. Available: https://blog.torproject.org/blog/iran-blocks-tor-tor-releases-same-day-fix 4

[11] Tor Project. (n.d.) obfsproxy. [Online]. Available: https://www.torproject.org/projects/obfsproxy 4

[12] ——. (2012, Nov.) Ticket #7520: Design and implement a social distributor for bridgedb. [Online]. Available: https://trac.torproject.org/projects/tor/ticket/7520 5

[13] D. McCoy, J. A. Morales, and K. Levchenko, "Proximax: A measurement based system for proxies dissemination," *Financial Cryptography and Data Security*, 2011. 5, 9, 10, 11

[14] Q. Wang, Z. Lin, N. Borisov, and N. J. Hopper, "rbridge: User reputation based tor bridge distribution with privacy preservation," *20th Annual Network & Distributed System Security Symposium*, 2013. 5, 9, 10, 11, 12, 20, 27, 32, 33

[15] 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. 6

[16] Y. G. Desmedt, "Threshold cryptography," *European Transactions on Telecommunications*, vol. 5, no. 4, pp. 449–458, 1994. 6

[17] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Robust and efficient sharing of rsa functions," in *Advances in Cryptology CRYPTO96.* Springer, 1996, pp. 157–172. 6, 7, 15, 17

[18] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979. 6, 7

[19] A. De Santis, Y. Desmedt, Y. Frankel, and M. Yung, "How to share a function securely," in *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing.* ACM, 1994, pp. 522–533. 7

[20] T. Rabin, "A simplified approach to threshold and proactive rsa," in *Advances in Cryptology CRYPTO'98.* Springer, 1998, pp. 89–104. 7, 12, 14, 15, 16, 18

[21] D. Boneh and M. Franklin, "Efficient generation of shared rsa keys," *Advances in Cryptology CRYPTO'97*, pp. 425–439, 1997. 7, 15

[22] N. Gilboa, “Two party rsa key generation,” in *Advances in Cryptology CRYPTO’99*. Springer, 1999, pp. 785–786. 7

[23] Tor Project. (2013) Bridge distribution database. [Online]. Available: https://gitweb.torproject.org/bridgedb.git 31

[24] The Python Software Foundation. (2013) 17.3 ssl – tls/ssl wrapper for socket objects. [Online]. Available: http://docs.python.org/2/library/ssl.html 31

[25] D. Litzenberger. (2013) Pycrypto - the python cryptography toolkit. [Online]. Available: https://www.dlitz.net/software/pycrypto/ 31

[26] (2013) gmpy: Multiple-precision arithmetic for python. [Online]. Available: https://code.google.com/p/gmpy/ 32

[27] Free Software Foundation. (2013) The gnu multiple precision arithmetic library. [Online]. Available: http://gmplib.org/ 32

[28] R. Dingledine and N. Mathewson. (n.d.) Tor protocol specification. [Online]. Available: https://gitweb.torproject.org/torspec.git/blob/HEAD:/tor-spec.txt 32

[29] D. Harkins and D. Carrel. (1998) The internet key exchange (ike). [Online]. Available: http://www.ietf.org/rfc/rfc2409.txt 32

Appendix

Figure A.1 shows the set of clients at the end of the 720-day simulation with random selection of bridges upon client registration. Nodes with an S are seeds. Black nodes are malicious nodes. The colors of nodes with 3, 2, 1, and 0 unblocked bridges are, respectively, green, yellow, orange, and red. The method below was the random selection as used for *rBridge*. In other words, when a client joins, he is given 3 random bridge addresses.

Figure A.2 shows the set of clients at the end of the 720-day simulation with inherited random selection of bridges upon client registration. Nodes with an S are seeds. Black nodes are malicious nodes. The colors of nodes with 3, 2, 1, and 0 unblocked bridges are, respectively, green, yellow, orange, and red. The method below was the "inherited" random selection. Here, when a client joins, he is given bridge IP addresses based on his parent node, the client that invited him. If the parent has 3 unblocked bridges, the child gets 2 of those at random. If the parent has 2 unblocked bridges, the child gets 1. The remaining bridges are chosen at random.

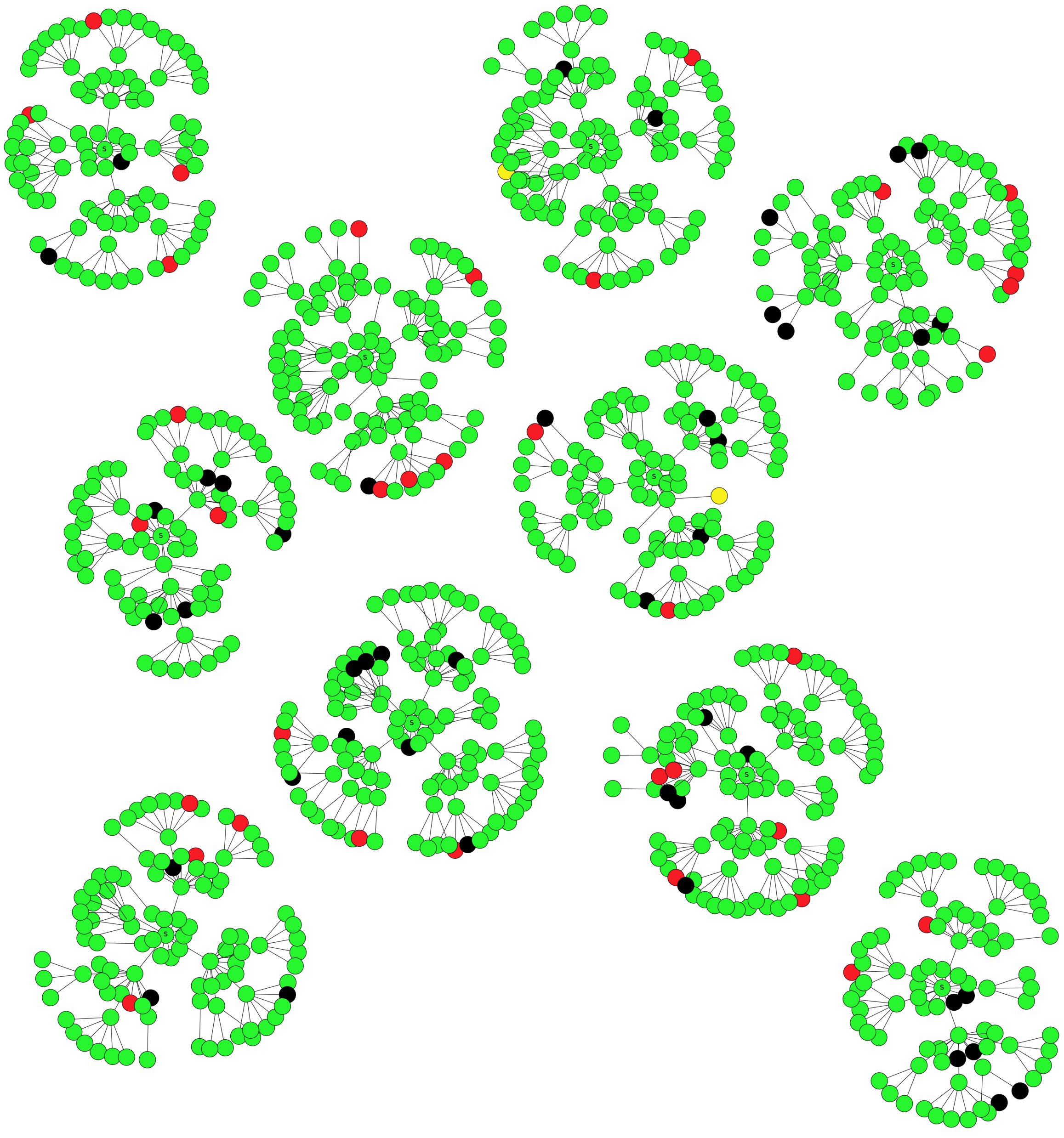

Figure A.1: Clients at the end of simulation with random bridge selection.

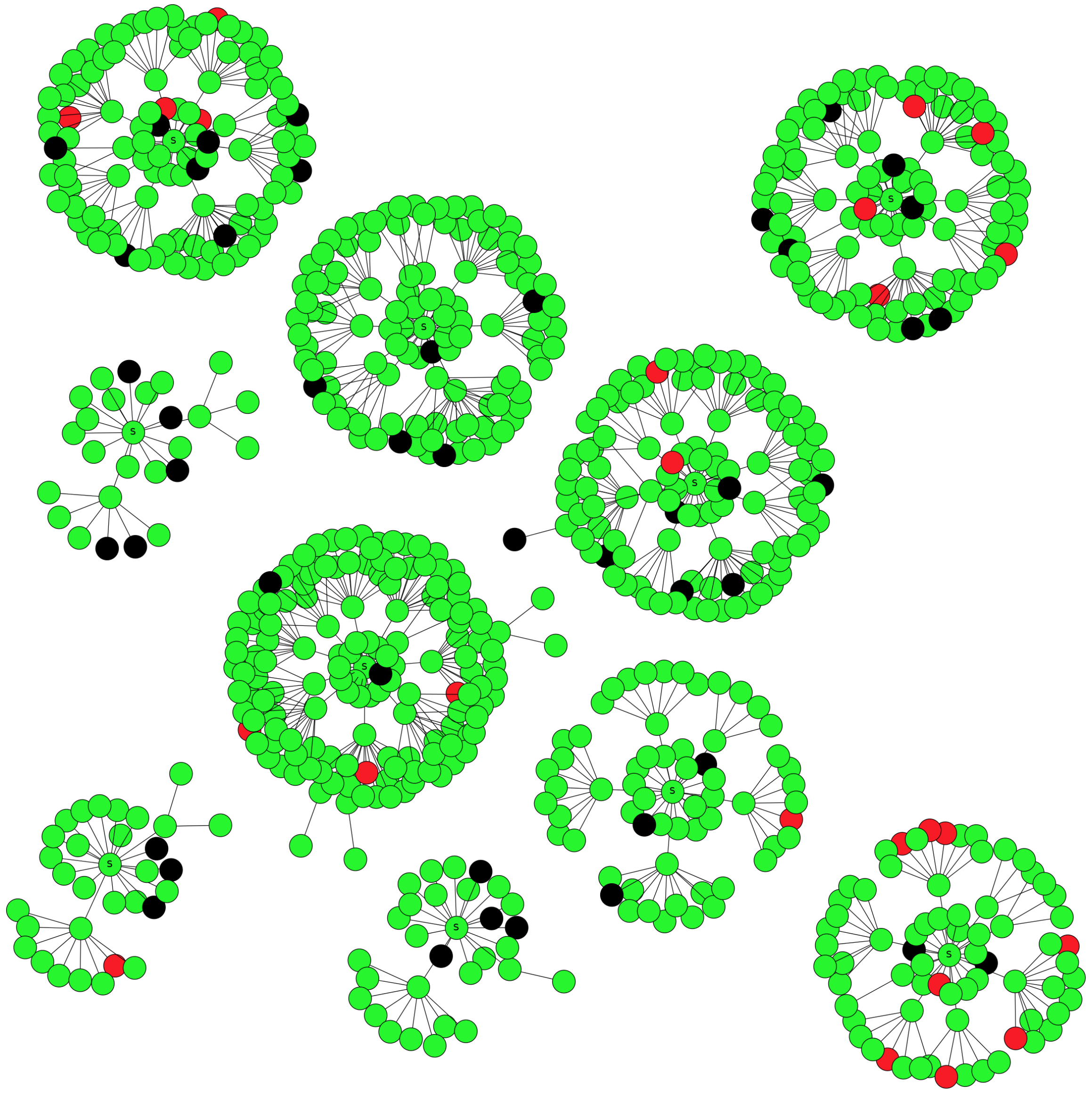

Figure A.2: Clients at the end of simulation with “inherited” random bridge selection.

Vita

Jordan Deyton was born in Bristol, TN to Rodney and Debra Deyton. He is the youngest of three brothers. He attended Greeneville High School in Greeneville, TN, from which he graduated as co-valedictorian in 2006. From 2006 to 2010, he attended the University of Tennessee in Knoxville, TN, where he earned a double major as a Bachelor of Science in Mathematics, Honors and Computer Science. His senior thesis, *User Interface for Manipulation of Möbius Transformations*, was co-directed by Dr. Brad Vander Zanden (EECS) and Dr. Ken Stephenson (Mathematics). After graduating *summa cum laude*, he entered the University of Tennessee's Electrical Engineering and Computer Science department as a graduate student. There, he studied under the advisory of Dr. Jinyuan Sun.

While an undergraduate of the University of Tennessee, Jordan worked for various employers. From 2007 to 2008, he worked as an undergraduate tutor for the Math Tutorial Center, part of the UT Department of Mathematics. In the fall of 2009, he was an undergraduate teaching assistant for Computer Science 140: Data Structures. From 2008 to 2010, he was a programmer and database manager for the Agricultural Policy Analysis Center of the UT Institute of Agriculture. Upon entering graduate studies with EECS in 2010 and since then, he has been providing departmental IT support as a member of the EECS IT staff.

Throughout his educational and professional experience, Jordan has developed a profound interest in cybersecurity, cryptography, computer networks, software engineering, web and GUI design, and databases. He has programmed in various languages, including C, C++, Java, Visual Basic.NET, Python, Perl, PHP, and shell scripting. Although he is familiar with OS X, his real affinity is for both Windows and Linux-based operating systems. Jordan aims to continue honing and expanding his interests and skills as he ends his academic career in 2013.