THE UNIVERSITY OF CALGARY

A Practical Buses Protocol for Anonymous Network Communication

by

Andreas Hirt

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

JUNE, 2004

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "A Practical Buses Protocol for Anonymous Network Communication" submitted by Andreas Hirt in partial fulfillment of the requirements for the degree of Master of Science.

Dr. Carey Williamson,
Supervisor,
Department of Computer Science.

Dr. John Aycock,
Internal Examiner,
Department of Computer Science.

Dr. Michael John Jacobson, Jr.,
Co-Supervisor,
Department of Computer Science.

Dr. Mark L. Bauer,
"Internal" External Examiner,
Department of Mathematics and Statistics.

Date

Abstract

The need to communicate anonymously over the Internet has increased with the proliferation of networked computers. Applications such as military communications, Web browsing, e-voting, and e-counseling for victims of abuse all require anonymous communication. Without anonymity, individuals may refrain from communicating for fear of retribution, potentially resulting in social, psychological, or financial losses, or even the loss of life.

This thesis contains a comprehensive survey and analysis of anonymous communication schemes. Analysis of the prior literature shows that there is no secure and scalable anonymous communication scheme. Previous literature has only analyzed each scheme for a subset of the known attacks. In this thesis, the analysis is extended to assess the anonymity capabilities of these schemes with respect to all known attacks. It is shown that none of the scalable anonymous communication schemes are secure.

The thesis contains a description of the design, implementation, and evaluation of a prototype anonymous communication scheme. The Buses anonymity protocol is identified as the most secure and scalable candidate protocol for a dynamic network topology. The protocol is re-designed and extended into the Practical Buses protocol, with features added to protect against all of the known attacks in the literature. New techniques are introduced to make the protocol scalable while preserving mutual anonymity. The design is extended to make the protocol more efficient, secure, and fault-tolerant. The experimental results obtained demonstrate that the Practical Buses protocol is a promising solution for anonymous network communication.

Acknowledgments

My sincere thanks to my supervisors Dr. Carey Williamson and Dr. Michael John Jacobson, Jr. They were always there to provide me guidance and support. I appreciated their prompt and constructive feedback and their help to improve my writing style. Dr. Williamson's knowledge with networking was a non-exhaustive source of information. His guidance on the structure of my thesis and implementation design issues of my protocol were invaluable. Dr. Jacobson's knowledge of cryptography ensured that I designed a protocol that was secure and efficient. Our discussions of how I was modifying the protocol were essential and a much needed reality check. The Practical Buses protocol would not have been designed had it not been for Dr. Jacobson's suggestion of pursuing nested encryption with indirection.

I wish to thank my thesis committee members: Dr. Mark Bauer and Dr. John Aycock for reviewing my thesis and for their valuable remarks. In addition, I would like to thank Dr. John Watrous for chairing my M.Sc. examination. My thanks also go to CISaC for providing the necessary infrastructure for me to carry out my experiments.

Most importantly, I would like to thank my wife Kristy and daughter Natalie Hirt. Their love, support, and belief in me never wavered. They provided a pillar of strength that nurtured the environment for me to complete my M.Sc. In addition, I wish to extend my gratitude to my family for believing in me and loving me: my parents Lothar and Christiane Hirt, my sister Nicola Focht, and my brother Tom Hirt.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

- ACI — Anonymous Connection Identifier
- ACM — Association for Computing Machinery
- Actv. Advsry. — Active Adversary
- AES — Advanced Encryption Standard
- Anon. — Anonymity
- Attk. — Attack
- Bf. — Before
- CA — Certification Authority
- CBC — Cipher Block Chaining
- Comp. — Compromised
- Corr. — Correlation
- DC-Net — Dining Cryptographer Network
- DoS — Denial Of Service
- Evdpr. — Eavesdropper
- GB — Gigabyte
- GMP — Gnu Multi-Precision Library
- HTTP — Hyper Text Transfer Protocol
- IEC — International Electrotechnical Commission
- IEEE — Institute of Electrical & Electronics Engineers
- IP — Internet Protocol
- ISO —International Organization for Standardization
- ISP — Internet Service Provider

- KB — Kilobyte
- KS — Key Seed
- LAN — Local Area Network
- LPWA — Lucent Personal Web Assistant
- Loc. — Local
- MAC — Message Authentication Code
- OAEP — Optimal Asymmetric Encryption Padding
- OFB — Output Feedback Mode
- PK — Public Key
- Pass. Advsry. — Passive Adversary
- Pxy. — Proxy
- RPI — Return Path Information
- RSA — Rivest, Shamir, & Adleman (public key cryptosystem)
- SHA — Secure Hash Algorithm
- SK — Symmetric Key
- SSA — Signature Scheme with Appendix
- TCP — Transmission Control Protocol
- Tb. — Traceback
- VPN — Virtual Private Network
- WAN — Wide Area Network

Chapter 1

Introduction

1.1 Motivation

The phenomenal growth of networked computers has fostered the deployment of many network applications: e-counseling for victims of abuse, command and control military communications, e-mail, Web browsing, e-voting, and e-shopping, to name a few. Many of these applications share private information between parties who may wish to conceal the identity of the sender, receiver, or both. The goal of an anonymous communication scheme is to keep the identities of communicating parties secret from eavesdroppers and adversaries.

Anonymous communication is necessary so that individuals can communicate without fear of retribution from an adversary or eavesdropper. Without anonymity, the lack of communication could have detrimental social and psychological effects in group therapy sessions or result in financial losses in government, business, and academic applications. In military applications, detection of any communication, or change in communication patterns could result in death on the battlefield due to an adversary gleaning information from communication patterns.

Some specific situations where anonymous communication is required are the following:

- Military communication should keep both the identity of the sender and receiver anonymous to prevent the enemy from gathering information from their communication patterns. In addition, the anonymous communication must be real-time in critical communications such as command and control on the battlefield.

- Group therapy sessions may have members that wish to keep their comments anonymous. In order to integrate these members, a dedicated network with an anonymous chat program could be used to keep the identity of the sender anonymous. As soon as a member feels comfortable enough to disclose their identity, the member could reveal their alias to trusted individuals.

- Government and businesses could use a dedicated anonymous network to allow employees to anonymously report sensitive information (whistle blowing) such as witnessed theft, scandal, et cetera.

- Group evaluations such as student reviews of a teaching assistant or professor could be done by using a dedicated anonymous network. The identity of the student would be kept anonymous to alleviate the student's fear of a negative response, and the identity of the party to whom the student is communicating could be authenticated.

The need for anonymity has motivated many different anonymous communication schemes: Chaum's Mixes [7], Onion routing [34, 41, 42, 43], Crowds [35], Hordes [38], Lucent Personalized Web Assistant (LPWA) [15], Anonymizer [2], Remailer [10] (types 0,1, and 2), Babel [18], Buses [4], DC-Net [8], Xor-trees [13], Peer-to-peer Personal Privacy Protocol (P^5) [37], Pipenet [30] and Freedom [14], some of which are surveyed in [6, 9, 17]. These anonymous communication schemes provide different kinds of anonymity, are meant for different application domains, utilize different techniques to provide anonymity, and have different strengths and weaknesses against a malicious attacker. However, it will be shown in Chapter 3 that none of these schemes are completely secure and scalable.

1.2 Types of Anonymity

There are two main types of anonymity [9], both of which must be provided in order to anonymize an application fully. *Data anonymity* is concerned with hiding the identities within the application data itself, and makes use of techniques such as remailers that strip all identifying information from email headers. *Connection anonymity*, on the other hand, is concerned with hiding identities at the network layer by obscuring communication patterns.

In this thesis, we are only concerned with connection anonymity. There are four types of connection anonymity, each of which may be required for various applications.

- *Sender anonymity* [29] keeps the identity of the sender anonymous. Sender anonymity is required for applications such as anonymous e-mail, e-voting, anonymous web-surfing, and e-counseling for victims of abuse.

- *Receiver anonymity* [29] keeps the identity of the receiver anonymous. Publishing documents on the web is one application in which receiver anonymity is often desirable.

- *Mutual anonymity* [17] provides both sender and receiver anonymity, and can be required for anonymous e-mail and anonymous bulletin boards.

- *Unlinkability* [29] means that an attacker cannot discover the identity of the sender, even if the identity of the receiver is known, or vice versa. Real-time communication between allied groups on a battlefield is one example where unlinkability is desirable, as an enemy can gain a tactical advantage by determining communication patterns.

Conceivably, there are additional types of anonymity. However, we restrict ourselves to the classical framework from the literature. Some of the other conceivable types of anonymity provided are addressed by resistance to various attacks in Chapter 3.

The type of connection anonymity provided by an anonymity scheme may be dependent upon perspective. For example, sender anonymity may be provided from the perspective of a receiver, but sender anonymity may be defeated by an eavesdropper that monitors the entire network. Also, no single kind of connection anonymity is universally the best; it is strictly dependent upon the application. For example, a "strong" form of anonymity such as mutual anonymity would be undesirable for an election, because a voter would want to be assured that the proper authority received his or her ballot.

Connection anonymity can be thought of as a continuum with different degrees of anonymity. The degree of anonymity can range from *absolute privacy* to *provably exposed*. The interested reader is referred to [35, 38] for more details on degrees of anonymity.

1.3 Thesis Objectives

The first objective is to survey the anonymous communication schemes from the literature, summarize the characteristics of these anonymous communication schemes, and analyze each scheme for a large set of defined attacks. Previous literature has only analyzed each anonymous communication scheme for a subset of these attacks; this analysis is extended and provides a complete picture of the anonymity capabilities of these schemes with respect to all known attacks.

The second objective is to re-design the original Buses protocol [4] in order to provide mutual anonymity and preserve scalability. This new design addresses several newly-discovered vulnerabilities and makes it fit for implementation. To achieve this objective, pre-existing and new techniques are incorporated into the re-designed Buses protocol.

The third objective is to implement a proof-of-concept prototype of the Practical Buses protocol, experimentally evaluate its performance, and assess its anonymity. The experimental and anonymity analysis demonstrate that the Practical Buses protocol is a promising secure protocol that is scalable, and has manageable overhead.

1.4 Thesis Contributions

There are four main contributions in this thesis:

- The thesis contains a comprehensive survey and analysis of anonymous communication schemes [19]. Analysis shows that all of the schemes except DC-Net are vulnerable to at least one attack. Several attack susceptibilities not previously published in the literature are found. In addition, a repository of anonymity attacks gathered from throughout the literature are brought together and grouped according to attack resources. The attributes needed to defend against each attack are also identified.

- The attributes needed to design a secure anonymous communication scheme are identified for the mixing, broadcasting, and buses anonymity techniques. From these required attribute lists, a software engineer could choose the appropriate anonymity technique for the application domain and design a secure anonymous communication protocol.

- The Buses protocol [4] is re-designed into the Practical Buses protocol. The new protocol is secure against all of the surveyed attacks, provides support for a dynamic network topology while preserving mutual anonymity, is scalable, and improves fault tolerance, security, and performance. To make the protocol resist all of the pre-existing attacks, replay protection is added. In addition, the new techniques *nested encryption with indirection*, *p-threshold replacement back-off scheme*, and *randomly delayed seat deletion* are introduced to provide a scalable framework that provides mutual anonymity within a dynamic network. Acknowledgments and signatures are added to improve the protocol's fault tolerance and awareness of active threats. Lastly, hybrid encryption, resends, dynamic bus size, seat expiration, and reduced seat decryptions are all added to improve performance.

- A proof-of-concept prototype of the Practical Buses protocol is implemented and evaluated [20]. To the best of the author's knowledge, this is the first implementation of *any* Buses protocol. Experimental analysis of the Practical

Buses protocol confirms that the protocol is scalable, and demonstrates its practicality for real world use.

1.5 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 contains a survey of the literature's anonymous communication schemes. Chapter 3 consists of a summary of the attacks in the literature and an analysis of each anonymous communication scheme for these attacks. A Practical Buses protocol is introduced in Chapter 4. The design and implementation of a Practical Buses prototype is explained in Chapter 5 and evaluated in Chapter 6. The evaluation consists of a performance analysis from experimental results and a theoretical anonymity analysis. Lastly, Chapter 7 contains a summary of the thesis and a discussion of future work.

Chapter 2

Background and Literature Review

This chapter contains the background information on anonymous communication schemes. The four main anonymous communication techniques used to provide connection anonymity are identified in Section 2.1. The anonymous communication schemes Chaum's Mixes [7], Onion routing [34, 41, 42, 43], Crowds [35], Hordes [38], Lucent Personalized Web Assistant (LPWA) [15], Anonymizer [2], Remailer [10] (types 0,1, and 2), Babel [18], Buses [4], DC-Net [8], Xor-trees [13], Peer-to-peer Personal Privacy Protocol (P^5) [37], Pipenet [30] and Freedom [14] are surveyed in Sections 2.2 to 2.13. In Section 2.14 the attributes utilized to provide anonymity for each surveyed anonymous communication technique are summarized. For clarity, examples of the four main anonymity techniques are provided in the Appendices A.1, A.2, A.3 and A.4.

Throughout the chapter, the following notation is used. S is the sender of a message M, and R is the receiver of the message. R's reply to the message M, is denoted M'. A symmetric key is used by a symmetric cipher such as AES and is denoted by K. Public keys and private keys are used by asymmetric ciphers such as RSA and are denoted by K^+ and K^- respectively. A subscript is appended to a key to denote the owner of the key. For example, $K_{(R,S)}$ is the symmetric key that belongs to R and S, and K_R^+ is R's public key. Encryption is denoted by E and decryption is denoted by D. For example, $E_{K_R^+}(M)$ is public-key encryption of S's message M with R's public key, $D_{K_R^-}(M)$ is private key decryption of S's message M with R's private key, and $E_{K_{(R,S)}}(M)$ is symmetric key encryption of S's message M with the symmetric key shared by R and S. Encryption of multiple arguments separated by commas denotes encryption of the arguments concatenated. For example, $E_{K_R^+}(M, K_{S,R}))$ denotes public key encryption with R's public key of the concatenation of S's message M with the symmetric key shared by R and S. Lastly, A denotes an address and a subscript is appended to denote who the address belongs to. For example, A_R is R's address and A_{mix_1} is the address of mix_1.

2.1 Anonymous Communication Techniques

To date, four techniques are used to provide anonymity: mixes, re-routing, buses, and broadcasting.

1. *Mixes* provide anonymity by re-routing a message through intermediate nodes, layering encryption to hide the contents of a message, and having intermediate nodes obfuscate the inputs with the outputs (see Section 2.2 for detailed explanation of Mixes). Mixes provide strong anonymity guarantees against a large array of attacks with the cost of an expensive overhead.

2. *Re-routing* re-routes messages through intermediate nodes and may use encryption. They are susceptible to a larger array of attacks than mixes, but are faster so that the end-users experience a smaller delay.

3. *Buses* uses layered encryption, along with the metaphor of a city bus which has a scheduled route through the network. The bus hides the message's route

through the network just like a public transit bus hides a passenger's route through a city (see Section 2.10 for detailed explanation of buses). Buses provides strong anonymity because it is secure against all attacks except replay attacks. Also, buses is scalable and the amount of overhead introduced is manageable.

4. *Broadcasting* combines broadcasting and encryption. If a message is encrypted and broadcast to a group, everyone in the group is equally likely to be the receiver and only the group member who holds the decryption key knows that it is the recipient of a message. Strong anonymity is provided, but these schemes are not scalable and have a large overhead.

2.2 Chaum's Mixes

The first paper on anonymous communication was written in 1981 by Chaum [7]. This paper introduces sender anonymous e-mail and e-voting. The key to anonymous communication is provided by a *mix*, an intermediate node in the re-routing path between the sender and receiver that uses layered public-key encryption and obfuscates incoming messages with outgoing messages. Chaum's mixes are primarily of theoretical interest, since the constant rate traffic overhead limits practical scalability. Weaker versions of Chaum's mixes that do not use constant traffic are used as a basis in Onion Routing, Mixmaster, and Babel. Chaum's mixes provide strong sender anonymity and unlinkability.

To prevent a mix from becoming a single attack point, a sequence of mixes called a *cascade* is used. A sender requires the public keys and addresses of all the mixes and the recipient in the re-routing path before it can send a message. In addition to a cascade, a certification authority (CA) is required to provide bindings between public keys and their respective identities.

When a mix collects enough messages, it sends all messages out in lexicographic order in a batch (lexicographic batching) to prevent timing correlations between input messages and output messages. In order to thwart traffic analysis, each sender must send the same number of messages in a certain time interval; hence, some of the messages are dummy messages. Uniform message sizes are used so that an attacker cannot determine size correlation between input messages and output messages of a mix. The initial message is padded with random bits to achieve the target message size. As a message passes through a mix, it shrinks and additional random bits are appended to the message to replace the lost bits. Keeping messages a constant size is implicitly assumed for the rest of the discussion on Chaum's mixes.

For a sender to send a message M to receiver R through mix$_1$ it sends:

$$E_{K_1^+}(R_1, E_{K_R^+}(R_0, M), R) \tag{2.1}$$

where $E_{K_1^+}$ denotes public-key encryption for mix$_1$ using the public key K_1^+, R_1 and R_0 are random bit-strings used for both random salt (randomizes encryptions with same content) and receipts, and $E_{K_R^+}$ denotes public-key encryption for the receiver using the public key K_R^+. After receiving the message specified in Equation 2.1, mix$_1$ decrypts the message with its private key, throws away the random salt R_1, determines the recipient's address R, and sends $E_{K_R^+}(R_0, M)$ to R. The receiver R then decrypts the message with its private key and throws away the random salt R_0.

To ensure that a mix outputs an item properly, a sender can request a receipt. In the above example of a single mix, the sender receives a receipt $Y = D_{K_1^-}(C, E_{K_1^+}(R_1, E_{K_R^+}(R_0, M), R))$ from mix_1 where C is a large constant known by everyone and $D_{K_1^-}$ is the private key decryption for mix_1 using the private key K_1^-. C ensures that mix_1 produced the signature by forcing $E_{K_1^+}(Y)$ to start with C. Otherwise, a malicious adversary could spoof mix_1 by replacing Y with random data. When mix_1 outputs multiple messages per batch, it signs the entire output batch as a whole and produces one receipt Y.

In the event of a dispute, the sender can provide Y, $X = (E_{K_R^+}(R_0, M), R)$, and R_1 to an arbitrator. To determine if the signature Y is correct, the arbitrator calculates $E^+_{K_1}(R_1, X)$ and checks if this appears in $E_{K_1^+}(Y)$. If the signature is incorrect, the arbitrator determines that mix_1 failed to output the message properly. Otherwise, the arbitrator checks if the correct message $E_{K_R^+}(R_0, M)$ was sent to the receiver ($E_{K_R^+}(R_0, M)$ is extracted from X). If $E_{K_R^+}(R_0, M)$ is not in the output batch of mix_1, then the arbitrator determines that mix_1 failed to output the message properly. If the signature and forwarded message are correct, then the arbitrator rules that mix_1 output the message correctly.

In order to send a message to R through a cascade of mixes $\text{mix}_n, \text{mix}_{n-1}, \ldots,$ mix_1, the structure in Equation 2.1 is applied recursively. That is, the sender sends to mix_n

$$E_{K_n^+}(R_n, E_{K_{n-1}^+}(R_{n-1}, \ldots, E_{K_2^+}(R_2, E_{K_1^+}(R_1, E_{K_R^+}(R_0, M), R), A_1) \ldots), A_{n-1}) \quad (2.2)$$

where $A_1, \ldots, A_{n-1}$ represent the addresses of $\text{mix}_1, \ldots, \text{mix}_{n-1}$, respectively. After receiving the message in Equation 2.2, mix_n decrypts the message with its private key, throws away the random salt R_n, and sends

$$E_{K_{n-1}^+}(R_{n-1}, \ldots, E_{K_2^+}(R_2, E_{K_1^+}(R_1, E_{K_R^+}(R_0, M), R), A_1) \ldots)$$

to mix_{n-1}. This process is repeated recursively by $\text{mix}_{n-1}, \text{mix}_{n-2}, \ldots, \text{mix}_1$ and the message $E_{K_R^+}(R_0, M)$ is ultimately received by the recipient and decrypted. Receipts can also be used in a cascade of mixes, but a node only needs the receipt of its successor node. The absence of an item from the predecessor's receipt can be used to substantiate the absence of an item from the node's input.

An extension to mixes provides an untraceable return address. A sender S wants to send a message to R with a return address $(E_{K_1^+}(R_1^+, A_S), K_S^+)$ where A_S is the sender's real address, K_S^+ is a public key generated for the reply by R, and R_1^+ is a public encryption key used by the mix on the reply. This return address R is appended to the message M, layered by encryption as described by Equation 2.1, and then sent to the receiver. R then sends the reply M' to mix_1 as $M'' = E_{K_1^+}(R_1^+, A_S), E_{K_S^+}(R_0, M')$. Then, mix_1 decrypts $E_{K_1^+}(R_1^+, A_S)$, extracts the address of the sender A_S, and sends $E_{R_1^+}(E_{K_S^+}(R_0, M'))$ to the sender S. R is the only one who can decrypt the initial message sent by S because it is the only one with the private key for K_R^+. Thus R is the only one who could know R_0 (see Equation 2.1) and this authenticates that R sent the reply message to S. Similarly, R_1^+ authenticates that the mix was the intermediate node in the path. Note that instead of using public key cryptography and the public keys K_S^+ and R_1^+, symmetric key cryptography could be used instead to improve performance.

An untraceable return address can be extended to a cascade. In this case, the return address that is appended to the receiver's reply message M' is

$$E_{K_1^+}(R_1^+, E_{K_2^+}(R_2^+, \ldots, E_{K_{n-1}^+}(R_{n-1}^+, E_{K_n^+}(R_n^+, A_S)) \ldots) A_2), E_{K_S^+}(R_0, M') \ .$$

The output of mix$_1$ that is sent to mix$_2$ is

$$E_{K_2^+}(R_2^+, \ldots, E_{K_{n-1}^+}(R_{n-1}^+, E_{K_n^+}(R_n^+, A_S)) \ldots), E_{R_1^+}(E_{K_S^+}(R_0, M'))$$

and the output of the last mix, mix$_n$, that is sent to S is

$$E_{R_n^+}(E_{R_{n-1}^+} \ldots E_{R_2^+}(E_{R_1^+}(E_{K_S^+}(R_0, M'))) \ldots) \ .$$

Another extension is *certified mail*. The last mix in the re-routing path that forwards $E_{K_R^+}(R_0, M)$ to the receiver R sends a receipt back to the sender which includes the message $E_{K_R^+}(R_0, M)$ and address A_R that it was delivered to, via an untraceable return address. The sender then verifies that the receipt is correct.

Mixes must defend against a *replay attack*. If an active adversary replays a message, then the adversary could eventually correlate the message with its output. For the same reason, return addresses cannot be repeated, and each time S wishes to receive a reply from R it must send a new return address, i.e., one that contains different public keys $R_n^+, R_{n-1}^+, \ldots, R_1^+$.

2.3 Onion Routing

Onion routing [34, 41, 42, 43] is an anonymous communication scheme written in 1997 that provides sender anonymity and unlinkability by using mixes. The mixes obfuscate the incoming messages with the outgoing messages by using *pseudo-random batching*. Each mix outputs the input messages in a pseudorandom order within a fixed interval of time. The metaphor of an onion is used because route information is layered with public key and symmetric key encryption in a similar manner as in Chaum's mixes. The outer layer of an onion is intended for the next hop in the re-routing path, and each hop removes the subsequent layer of the onion and forwards it. The onion is used to create a bi-directional anonymous path. This anonymous path can then be used by both the sender and receiver to exchange messages.

Onion routing provides sender anonymity and unlinkability and is available for general public use, see [28]. However, this implementation can only be used in conjunction with virtual private networks (VPNs), Web browsing, e-mail, remote login (rlogin), and electronic cash because it must be able to understand the application layer data it receives. For example, when remote login (rlogin) is used the rlogin command must make a connection to the application proxy (defined below). The application proxy must then be able to determine that the data received is an rlogin packet, parse the server and user name, and send the rlogin request through the anonymous Onion routing network. The last hop in the anonymous path must then be able to construct a valid rlogin request on behalf of the sender.

The infrastructure required for Onion routing is an onion proxy and a static Onion routing network. In remote proxy configuration, the sender requires an application proxy and the public key and address of an onion proxy. Onion routing requires a CA as a trusted authority.

In order to use Onion routing, clients adjust their application to make a socket connection to an application proxy running locally on the client's machine. Application independence is provided by the application proxy, which massages the application's message into formatted cells that the onion proxy can understand. First, a standard structure encrypted with the public key of the onion proxy is sent to the onion proxy. The standard structure contains the destination address, address format, and protocol used by the onion proxy. Then, the message is encrypted with the public key of the onion proxy and sent to the onion proxy.

When an onion proxy receives a standard structure, the onion proxy creates an *onion*. An onion is used to set up an anonymous connection for data transfer, but it does not contain the standard structure or messages. After an onion creates a static anonymous path, the standard structure is sent and thereafter messages can be sent using symmetric key encryption.

Each layer of the onion contains seven fields: Version, Back F, Forw F, Destination Port, Destination Address, Expiration Time, and Key Seed Material. The version specifies the version of Onion routing being used. Back F specifies the cryptographic function applied in the backward direction of data (receiver to sender) and Forw F specifies the cryptographic function to be applied in the forward direction of data (sender to receiver). The Destination Port and Destination Address fields specify the next onion router in the path. Expiration Time lets an onion router know how long it must guard against replay attacks before the path expires, and is specified in seconds relative to 00:00:00 UT January 1, 1970. The 128-bit long Key Seed Material is used to generate three symmetric keys: key_1, key_2, and key_3. key_i is generated by applying the SHA hash function to the key seed material i times. key_3 is used for the forward direction of data, key_2 is used for the backward direction of data, and key_1 is used in the generation of the next inner onion layer.

When preparing an onion, the onion proxy first determines the path of onion routers, $p_1, p_2, \ldots, p_k$. The onion is then constructed iteratively from the inner core outward. The inner core is destined for the last onion router in the path p_k, the next onion layer for p_{k-1}, and so on, with the outer onion layer for p_1. The inner core consists of 100 random bytes of data. Then, each 128-bit additional layer is prepended to the onion iteratively. The added layer is encrypted with the public key of the onion router for which the layer is intended. Each time a layer is added, the rest of the onion is encrypted with key_1 from the added layer using symmetric key encryption (DES OFB with an Initialization Vector of 0).

If an onion router receives an onion with Destination Address and Destination Port of 0, then it knows that it is the last hop in the path, and passes the connection to its own exit funnel. The exit funnel then knows that the standard structure follows, containing the receiver's destination address. It then attempts to set up a socket connection with the ultimate destination, retrying the specified maximum number of times in the standard structure before giving up. Any messages that the exit funnel forwards to the receiver are encrypted with the receiver's public key. Each individual onion router that receives an onion chooses a locally unique anonymous connection identifier (ACI). The onion router builds a routing table that maps the old ACI to the predecessor onion router, and the new ACI to the successor onion router. This makes it possible for messages to be sent in both directions in an established anonymous connection path.

Onion routing I [41, 42, 43] uses a core of 5 onion routers. Each hop is chosen at random by the onion proxy and all path lengths are 5. Onion routing II [43] uses a core of 50 Onion Routers. Each hop is chosen at random by the onion routers and the path length is random with a minimum length of 1. The path length is randomized as follows: each onion router flips a biased coin which is weighted with probability p_f. Let the weighted side of the coin be referred to as "heads". If the flip results in heads, the onion is forwarded to another onion router. If the flip is tails, the anonymous path ends and a socket connection to the receiver is made. The router then waits for the standard structure.

2.4 Crowds

Crowds [35] utilizes the idea of "blending into a crowd", a group of geographically diverse members, and was written in 1998 to provide anonymous Web surfing. Any crowd member can issue a Web request on behalf of another user. Sender anonymity is preserved because, to the Web server (the receiver), the request is equally likely to have originated from any member (the sender) in the crowd. Crowds utilizes re-routing with path encryption. *Path encryption* uses one symmetric key to encrypt the message at the beginning of the re-routing path and the same symmetric key to decrypt the message at the end of the re-routing path.

Crowds provides sender anonymity and unlinkability. It is used in practice [11] in conjunction with Web browsing. In order to use crowds, a user downloads and installs the software from [11] and changes their browser's proxy settings.

The infrastructure of Crowds consists of two components: Jondoes and a Blender. A Blender is a centralized trusted server that Crowds requires to maintain crowd memberships. A user must establish an account, account name and password with the Blender. Then, when a user starts a Jondoe, a locally running proxy, it contacts the Blender and authenticates itself with the username and password so it can become a member of a crowd. If the Blender authenticates the user, it adds the user's IP

Address, port number, and account name to the list of crowd members. In addition, the Blender generates a list of shared symmetric keys, one for every other member in the crowd, so that the user can exchange authenticated and encrypted messages with each crowd member. This list is sent to the user encrypted under the account password. Also, each newly generated key is appropriately sent to each other member of the crowd encrypted under their respective account password, informing them of the new crowd member's IP address, port number, and account name.

After joining a crowd, a single static re-routing path with a minimum length of 1 is created for all the user's communications. The user's Jondoe chooses a random Jondoe in the Crowd to be the first hop in the re-routing path. Then the user sends the re-routing path creation request to the first hop's Jondoe. Thereafter, any Jondoe that receives the path creation request flips a biased coin. With probability p_f, the probability of forwarding, the Jondoe randomly chooses another crowd member to which the path creation request is forwarded. In order to route the messages, a path ID is used as a tag on the message to determine the next hop. When creating a path, a Jondoe replaces the path ID with a new path ID, a randomly chosen 128-bit integer. The old and new path ID, along with the IP address and port number of the predecessor and successor nodes are recorded in a routing table. Replacing the path ID allows for the same Jondoe to appear in the path multiple times, but still route a message differently each time it appears on the path. If the flip of the biased coin indicates not to forward the message, it is then sent directly to the server. The first message sent by a new Jondoe utilizes this path creation, and every subsequent message follows this same path regardless of the ultimate Web server destination.

Re-routing paths are static because dynamic paths reduce sender anonymity (see

Theorem 5.2 in [35]). If Jondoe A receives a message from Jondoe B, all Jondoes excluding A and B are equally likely to be the sender, but B is most likely to be the sender. If re-routing paths are dynamic, then every time a Jondoe sends a new message a new path is created. If the next hop Jondoes in each of these paths collaborate, more messages being sent by the sender may compromise the anonymity of the sender.

On the other hand, with static path creation, any new user who joins a crowd and creates a new path is easily identified as a sender of a message. To solve this problem, a join commit is introduced. Periodically the Blender sends a join commit to all members of a crowd, and all of the crowd members at the same time create a new path. A new user refrains from creating a path when the user first joins, and waits until they receive a join commit.

To reduce latency, a *path key* is introduced. A path key is a symmetric key generated by an initiating Jondoe and is forwarded to all Jondoes on the path via the shared symmetric keys. This allows the use of path encryption for messages and replies. The exchange of the path key is done securely with the symmetric keys given by the Blender.

Crowds also prevents a *timing attack* (defined in Section 3.2). When Web browsing, a user may request a Web page with embedded objects. When the user receives the Web page the browser automatically requests the embedded objects. If the first hop in the anonymous path notices a sufficiently short time between sending the response and additional anonymous messages, then with high probability the first hop identifies the sender. To prevent this attack, Crowds has the last hop in the anonymous path check to see whether there are embedded objects, and if so, it requests them on behalf of the user that initiated the request. The user's Web browser is also changed to wait for embedded objects rather than request them itself.

2.5 Hordes

Hordes [38] provides sender anonymity and unlinkability for Web browsing by using re-routing, link encryption, and multi-casting and was written in 2000. In order to use Hordes, a client runs a local process h that is analogous to a Jondoe in Crowds. Hordes utilizes similar techniques as Crowds for sending a message to a receiver, but uses multicast on a receiver's response while still maintaining sender anonymity. Multicast reduces the round trip time to about half of that in Crowds because multicast is more direct (Crowds has longer return paths due to re-routing). The infrastructure in Hordes consists of a Blender and CA as trusted authorities, and Jondoes (the clients running process h). The Blender and Jondoe protocols are slightly different from Crowds. Hordes has not been used in practice.

Hordes consists of two main steps: initialization and data transmission. Initialization provides the sender with a list of all other Hordes members and their public keys as well as a multicast class D address A_g (a class D address is the type of IP address used for multicasting). This process is very similar to Crowds, except Hordes uses nonces (a value used only once) to ensure fresh lists are received, and timestamps to ensure that membership updates are not replays.

Data transmission in Hordes on the forward path from sender to receiver has three main differences from Crowds. Senders must randomly pick a multicast class D address A_g from A_G, the set of multicast class D addresses shared by all Jondoes in the Horde. Also, senders must generate a random 128-bit id, which allows a sender to determine if a received multicast is a reply to one of its messages. Both A_g and id are sent along with the message so that the receiver can respond to a sender by multicasting to A_g. Lastly, the major change is that Hordes does not use static re-routing paths. Instead, each hop in the re-routing path chooses a subset b of all Horde members to which a message is forwarded. It then exchanges a symmetric key k_f to be used for symmetric key encryption with all members of b. k_f is encrypted with the next hop's public key and the current node's private key. To determine if the message is forwarded to another Jondoe, a biased coin weighted with probability p_f is flipped. p_f is a configurable parameter that gives the probability of forwarding the message to another Jondoe and as p_f increases, the average path length increases. If the coin flip results in forwarding data, a random member of b is chosen by the Jondoe and selected as the next hop, after which the message is forwarded using symmetric key encryption with k_f. Otherwise, the coin flip results in the message being sent to the receiver. The use of dynamic paths in Hordes seems as if it would compromise the anonymity of the sender, but in fact "the size of the subset is chosen so that the number of forwarders in the set is on the order of the expected number of paths a Jondoe would be on in Crowds" [38, page 38].

Data transmission in Hordes on the backward path from receiver to sender is quite simple. To send a message, the id is appended to the front of the message, and the message is multicast to A_g. A subset of all Horde members, A_g, is chosen because this allows a sender to listen to a subset of all replies for the entire group A_G. If a subset is not chosen, a sender could waste a lot of resources listening to broadcast replies that are not intended for it. This drastically reduces the time for a reply from a receiver because the reply is sent directly via multicast versus taking extra re-routing hops.

Multicasting uses class D addresses. A group of members subscribe to a class D address. This effectively creates a tree of routers, and no individual router knows the entire membership. To know all the members of the group A_g requires the co-operation of all the routers and even then, each member of A_g is equally likely to be the recipient.

2.6 Lucent Personal Web Assistant

On the Internet, clients commonly type in login names and passwords that allow the user to be identified, sacrificing data anonymity. Lucent Personal Web Assistant (LPWA) [15] provides *weak sender anonymity* by using re-routing with no encryption and data anonymity of login names, secret key or password, and HTTP headers. Sender anonymity is weak because the re-routing path only consists of the LPWA proxy, which is easily defeated by an eavesdropper between the sender and the LPWA proxy. LPWA also prevents servers from establishing a *dossier*, the set of servers the client has interacted with so far.

LPWA can be used in conjunction with e-mail and Web browsing and was used in practice in 1997. It was then the basis for ProxyMate which was sold to CMGI's NaviPath in May 2000 [24]. NaviPath has not released its product yet [31]. The infrastructure of LPWA consists of a single LPWA proxy as a trusted authority.

A sender passes a single passphrase to the secure Janus function which provides the sender with a persona for the rest of the session with a particular server. The persona includes an alias user/login name, an alias secret key or password, and an alias e-mail.

In order to use LPWA, clients configure their browser settings to use the LPWA proxy, such as lpwa.com. The LPWA proxy is located at the edge of the subnet of clients, usually at an Internet Service Provider (ISP) or where a corporate firewall is located. The proxy utilizes filters and the Janus engine, a software implementation of the Janus function, to provide anonymity. It is assumed that the communication between the sender and the LPWA proxy is secure and the passphrase is sent as plaintext (not encrypted text) to the LPWA proxy. The LPWA proxy is the one intermediate node in the re-routing path.

The keys to data anonymity in LPWA are filtering and the Janus engine residing on the LPWA proxy. Filtering removes the From and Referrer headers from HTTP requests and trims the User-Agent header to remove platform specific information. The From header can contain the user's e-mail address. The User-Agent header contains information about what type of machine the user has. The Referrer header contains the URL of the Web page in which the URL of the current request appeared.

The purpose of the Janus function is to provide a client with consistent personas, such as aliases for HTTP login pages, while preserving the anonymity of the client. The Janus function J is defined as $J(c_i, s_j, p_i, t) = MAC_{p_i}(c_i||s_j||t)$ where MAC_{p_i} is the message authentication code generated by a CBC-MAC such as Advanced Encryption Standard using p_i as the key. In this notation c_i is the client, s_j is the server, p_i is the passphrase (the secret), and $t \in \{u, p, m\}$ denotes whether the client wants an alias username, password, or e-mail address, respectively.

The Janus function protects against collaborating servers establishing a dossier for a client by providing data anonymity. Collaborating servers obtain a unique alias (login name, password) tuple for each server because the Janus output is dependent upon s_j. The following secrecy and anonymity properties of the Janus function [15] ensure that collaborating servers cannot establish a dossier. The secrecy property prevents the collaborating servers from deriving the results of the Janus function from other results. The anonymity property prevents the collaborating servers from identifying who the client is given pairs of (alias,password) from different clients. The only other way for servers to establish a dossier is to analyze the referrer header in a HTTP request. However, the Janus engine removes the referrer header and a server cannot establish a dossier this way either.

To formally define the secrecy and anonymity properties, we first introduce some notation as in [15]. Let $a^u_{i,j}$ denote the alias username for client c_i to server s_j and $a^p_{i,j}$ denote the corresponding password for client c_i to server s_j. A client c_i is uncorrupted if the adversary is unable to determine p_i. A client c_i is opened to server s_j if the pair $(a^u_{i,j}, a^p_{i,j})$ was communicated to the server. A client c_i is identifiably opened to server s_j if the client is opened and the server can determine the corresponding c_i. Now, let us take the definitions of the secrecy and anonymity properties directly from [15, page 397]:

1. *Secrecy*: Given a server s_j, an uncorrupted and not identifiably opened client c_i, and $t \in \{p, u, m\}$, the adversary E cannot find $J(c_i, s_j, p_i, t)$ with non-negligible probability even under a chosen message attack, under the assumption that the adversary can get $J(c_i, s_{j'}, p_i, t')$ for any $s_{j'} \neq s_j$ or $t \neq t'$.

2. *Anonymity*: Given a server s_j, and two uncorrupted clients c_i, $c_{i'}$ not opened

with respect to s_j, and $t \in \{p, u\}$, an adversary cannot distinguish the value of $J(c_i, s_j, p_i, t)$ from $J(c_{i'}, s_j, p_{i'}, t)$ with non-negligible probability even under a chosen message attack, under the assumption that the adversary can get $J(c_{i''}, s_j, p_{i''}, t)$ for any list of arguments not used above.

The work in [15] proves that J satisfies these properties.

Weak sender anonymity for e-mail is also provided. In order to make LPWA transparent to the user, e-mail forwarding is used. The forwarding engine encrypts the user's e-mail address. Suppose the result of the encryption is X. The forwarding engine then replaces the e-mail's reply address with X@lpwa.com. When the server responds to this alias e-mail address, the forwarding engine decrypts X and forwards the e-mail to the decrypted address. The documentation in [15] does not specify whether public key or symmetric encryption is used. However, since only the LPWA proxy needs to store the encryption/decryption key, symmetric encryption is probably used since it is faster.

One attack to which LPWA is susceptible is if an attacker collects outputs of the Janus function, hoping for an internal collision (a birthday attack). The attacker would replace the first n bits of a message with alternatives, until one of the alternatives yields the same Janus function output as the original message. Using this method, a consequence of [5] shows that an attacker has to try approximately $2^{(keysize)/2}$ samples. Thus, 64-bit keys are insecure and keys of at least 128-bits are recommended [15].

2.7 Anonymizer

Similar to LPWA, Anonymizer [2] was written in 1997 to provide weak sender anonymity for Web browsing by using re-routing with no encryption. Only one intermediate node is used in the re-routing path, namely the Anonymizer server. A client changes the browser proxy to the Anonymizer server, and all subsequent messages are re-routed through the Anonymizer server. Anonymizer strips off all identifying headers and source addresses of the HTTP request. It then forwards the request on behalf of the client to the server. All responses are sent back to the Anonymizer Server, which forwards them back to the original sender. Thus, a server can only learn the identity of the Anonymizer server. Anonymizer requires the Anonymizer proxy as a trusted authority.

Currently, Anonymizer is available at [2]. In order to use Anonymizer, a user must pay a fee to create an account with an Anonymizer server and change their web browser's proxy settings.

2.8 Remailers

The purpose of remailers [10] is to provide sender anonymity for e-mail. The first remailer written was the type 0 remailer Anon.penet.fi in 1993. Type 1 and type 2 remailers also provide unlinkability. Only type 1 remailers allow a receiver to reply anonymously.

A type 0 remailer is based on re-routing with no encryption. It is very similar to Anonymizer. The required infrastructure is a single proxy as a trusted authority that provides one intermediate node in the re-routing path. The sender sends e-mail to the intermediate proxy, which strips all identifying headers from the message and forwards it to the receiver. The intermediate proxy and the connection from the sender to the intermediate proxy are assumed to be secure for sender anonymity to be preserved. Currently, there are many type 0 remailers used in practice, see [3] as an example. However, due to the amount of spam generated by type 0 remailers, their existence is usually short-lived.

A type 1 remailer is commonly referred to as a *Cypherpunk remailer*. The infrastructure required is a cascade of weak mixes. The mixes are weak because they rely on message delay, which only partially obfuscates inputs with the outputs of a mix. In addition, the weak mixes do not use uniform message size which permits a size correlation attack because each layer of encryption is removed at each weak mix without adding padding before forwarding the message. Type 1 remailers require a certification authority (CA) as a trusted server. Currently, there are many type 1 remailers used in practice, see [44, 45] for an extensive list.

An e-mail in a type 1 remailer system is conceptually a layered public-key encrypted structure similar to an onion. The inner core of the layered structure contains the message. Each layer contains routing instructions for each mix: delay, next hop, and optionally an alias. The delay instructs a mix to hold a message for *delay* time units, to thwart traffic analysis. The next hop (IP address) tells the mix whether it is to send the inner layers to the next mix or to the receiver. The alias allows the receiver to respond to an e-mail, and each mix maps the alias address to the predecessor node from which it received the onion. Cypherpunk remailers also strip off all (or at least some) mail headers and add new headers to provide data anonymity. Lastly, instead of specifying a delay, some type 1 remailers use a *message pool*. A message pool maintains a minimum number of messages in the pool; once the message pool is full an incoming message causes a message to leave the pool. To prevent an attacker from correlating an incoming message with an outgoing message, the messages in the pool are reordered with each incoming message. To send a message, a sender requires the public keys and addresses of all the mixes.

A type 2 remailer, *Mixmaster*, attempts to eliminate attacks that can be used against Cypherpunk remailers (see Section 3.3.1) by using mixes that delay and reorder messages, with uniform message size, protection against replays, and the absence of a means for the receiver to reply. The origin of its name comes from the fact that it is strongly based upon Chaum's mixes. The body of the message is 10 kilobytes, and the message is sent through an infrastructure of 20 mixes. The sender requires the public keys and addresses of 20 mixes. Mixmaster requires a CA as a trusted server. Currently, there are some Mixmasters used in practice, see [44].

Each layer of the message is encrypted with the public key of the mix for which it is intended. A received message is decrypted with the private key of the mix, revealing the header, which contains the next hop, a packet ID, and a symmetric triple DES key (3DES). The mix checks to make sure it has not seen the packet ID before, discarding if necessary to prevent replays. The next hop is extracted as well as the 3DES key. Then, the header is moved to the end of the list of headers (before the body of the message), and the entire message is decrypted with the 3DES key. The packet is then forwarded to the next hop. Flags in the core of the message indicate if it is the last hop and if it is a multi-part message. The body of the message is then placed into a reordering pool. If it is part of a multi-part message, the last hop does not place it in the pool until all other parts have arrived. The last hop identifies these multi-part messages with a flag, as well as the messages having the same packet ID but different Message IDs. All parts of the multi-part message must be received within a certain time limit, otherwise the entire message is discarded. How the multi-part message is re-assembled in the correct order is not specified in [10]. However, the problem is easily solved by adding sequence numbers to multi-part messages.

Mixmaster does not provide perfect sender anonymity if the traffic on the network is low. Correlations can be discovered between a sender and a receiver receiving a message a certain time period later. To defeat traffic analysis, dummy messages would have to be created to maintain a minimum level of traffic.

2.9 Babel

Babel [18] provides sender anonymity, unlinkability, and optionally receiver anonymity for e-mail and was written in 1996. It utilizes a cascade of mixes that use interval batching, probabilistic deferment, inter-mix detours, and fixed-set random order paths (as defined below) to obfuscate their input messages with their output messages. It enables stateless remailers by allowing a sender to append an optional untraceable return address (as defined below). Babel has not been implemented.

In order to use Babel a user either installs a filter in the .forward file (Unix configuration file for e-mail forwarding) or changes the proxy to the Babel Proxy. When a .forward file is used to install Babel, a password that is needed as a secret key is stored in plaintext. When using the proxy mode of operation, the user's password is sent to the proxy mix in plaintext. In proxy mode, it is recommended that

PGP [47] be used in conjunction with Babel to protect the password. In addition, it is recommended to use PGP for the original message that is sent to the receiver to hide the message contents during the first hop (proxy mode) and last hop (both .forward and proxy mode).

The infrastructure of Babel is a cascade of mixes and a CA as a trusted server. The sender requires the addresses and the public keys of the mixes, as well as a password (.forward configuration) or the first mix proxy's PGP public key (proxy configuration). Optionally, the PGP key of the receiver is needed.

The fundamental building block of Babel is a mix. To improve the chances that an adversary does not compromise the only intermediate mix, a cascade of mixes is used. Babel utilizes a number of techniques to obfuscate incoming messages with outgoing messages of a mix. *Uniform message length* prevents size correlation of incoming and outgoing messages. *Random one-time salt via symmetric session keys* ensures that should an eavesdropper intercept a message going out of a mix, they cannot apply the public key of the mix to discover what the incoming message was. These session keys also improve performance because symmetric key cryptography is computationally faster than public-key cryptography. A public key is used to encrypt the session key, and the session key is used to encrypt the rest of the message. *Interval batching* divides time into equal intervals T. If fewer than M messages are received within time T, decoy messages are generated so that the batch has M messages. A *variant of a timestamp* scheme uses the session key to uniquely identify messages and guard against replays since the chance of a collision of a randomly generated 128-bit integer is considered negligible. The mix must record these session keys with their timestamps, and can flush its memory of these after some fixed system-wide time interval.

The composition of a message for a sender is similar to an onion in Onion routing. However, the message is padded to exactly Ω bytes, and after each layering and encryption, the message is trimmed to Ω bytes. Unlike onion routing, the encrypted layered message in Babel does not set up the symmetric keys for a route; rather, the inner core contains the message. The maximum message size is $\alpha < \Omega$, which ensures that trimming does not remove the actual contents of the message. α is dependent on the number of hops in the re-routing path, how much data each layer adds, and Ω. Also, it is possible for the sender to give multiple mix addresses at each layer, which hides the next hop from the current processing mix. Only one mix from the multiple mixes is able to decrypt the message. Thus, a mix only knows the predecessor mix and not the successor mix. Lastly, the layer's header is encrypted with the public key of the mix the layer is intended for, and the rest of the message is encrypted with the symmetric key contained within the header.

The anonymous reply path is stored in return path information (RPI), allowing the mixes to be stateless. The sender randomly chooses a key seed (KS). Then, the sender generates r symmetric keys $K_1, K_2, \ldots, K_r$ using a hash function iteratively on KS, and generates a return path of mixes $R_1, R_2, \ldots, R_r$. The inner core of the RPI is $y_0 = E_{K_{S^+}}(KS, r)$, where $E_{K_{S^+}}$ is encryption with the sender's public key and r is the number of hops in the return path. The sender, starting with the last mix on the return path R_r calculates

$$y_i = (A_{R_{r-i+1}}, E_{K^+_{R_{r-i+1}}}(K_{r-i+1}, y_{i-1})) \text{ for } 1 \leq i \leq r \ .$$

The resulting RPI

$$y_r = A_{R_1}, E_{K^+_{R_1}}(K_1, A_{R_2}, E_{K^+_{R_2}}(K_2, \ldots, E_{K^+_{Rr}}(K_r, A_{sender}, E_{K^+_S}(KS, r)) \ldots))$$

is prepended to the beginning of the plaintext message. The sender sends this message with the prepended RPI to the receiver as an encrypted layered message. The last mix in the sender's re-routing path sees the RPI, and modifies the mail header so that the receiver replies to R_1.

To reply to an e-mail, the RPI is treated opaquely and prepended to the reply. This reply is sent to R_1, which is extracted from the mail header of the sender's message. Each mix R_i on the return path first decrypts the outer layer of y_i with its private key $K^-_{R_i}$ and extracts the symmetric key K_i and $A_{R_{i+1}}$ (the address of the next hop). It then prepends the RPI with w random bytes (w is the size of a layer) and deletes the first w bytes to keep the resultant RPI the same size. It applies the symmetric key K_i to the new RPI and the entire message. Ultimately, the last mix R_r sends to the sender $E_{K^+_S}(KS, r) +$ random data prepended to

$$Y_r = (E_{K_R}(E_{K_{R-1}}(\ldots E_{K_1}(\text{reply message}) \ldots))$$

The sender then applies the private key K^-_S to extract KS and r, which tells where the message is located (rw bytes later). The sender then generates the r symmetric keys from KS, which can be applied to Y_r to extract the reply message. Note that unlike Chaum's mixes, the prepended $E_{K^+_S}(KS, r)$ makes the sender almost stateless as well. The sender does not have to remember all the random keys generated for reply paths, just its own private decryption key K^-_S.

RPIs allow Babel to provide mutual anonymity. Alice can send an anonymous message to a newsgroup or bulletin board and include an RPI. Bob can then anonymously check the newsgroup or bulletin board, and send a reply to Alice. If Bob is concerned that Alice is trying to trick him into revealing his identity by controlling the RPI's return path, Bob can send an anonymous message to R_1 of Alice's RPI via an encrypted layered structure. Similarly, Bob can include an RPI and Alice can reply to Bob's reply via an anonymous message to R_1 of Bob's RPI via an encrypted layered structure. However, this allows for a replay attack, since Alice could receive multiple replies to her posted message.

Inter-Mix detours is a method used to prevent replay attacks and spam attacks. A mix R_i could send an anonymous message on the path $I_1^i, I_2^i, \ldots, I_{r_i}^i$. Then, $I_{r_i}^i$ can forward the message to R_{i+1}. An anonymous message on a detour is tagged to prevent infinite detours. A detour is quite simple, it is just an anonymous message that takes a detour created by a mix. Inter-Mix detours can be applied on the forward path, which has the nice property that even a sender cannot recognize its own message as it leaves a mix. Similarly, inter-mix detours can be applied on the reply path.

Additional techniques that are used to improve mixing are *probabilistic deferment* and *fixed-set random order path.* To provide maximal traffic going through the mixes, fixed-set random order path imposes that a path must pass through all mixes exactly once. To decrease the probability of an adversary correlating an incoming message with an outgoing message, probabilistic deferment is introduced. Probabilistic deferment implies that with a certain probability, a message may be deferred to the next outgoing batch.

2.10 Buses

The inspiration behind Buses [4] written in 2003, is the observation that public transit buses hide the movement patterns of passengers. In the networking context, the key idea is that messages can travel in the “seats” of a “bus” (a large packet with assigned message fields), appropriately obscured by symmetric key or public-key encryption. The bus traverses the network, passing from one node to the next. When in possession of the bus, a node can retrieve messages intended for it or place messages on the bus to be delivered to another node. By encrypting the messages, only the intended recipient can recover messages intended for it. If a semantically secure cryptosystem [26] is used, then an adversary cannot distinguish in polynomial time whether data on the bus is the encryption of a valid message or random data, and thus cannot even determine whether communication is taking place at all.

Buses [4] is used to send anonymous messages using two different techniques: reserved seats or layered encryption (defined below). Reserved seats does not provide sender anonymity from the perspective of the receiver, but from the perspective of everyone else mutual anonymity is provided. Layered encryption provides mutual anonymity from everyone’s perspective. Since Buses uses either symmetric or public-key encryption, we will use “encryption” to denote either symmetric or public-key encryption and “key” to denote either symmetric or public key. Buses is distinct from re-routing and mixing anonymity communication schemes because they do not require constant traffic to thwart traffic analysis nor do they require all processes to be busy constantly to provide anonymity. A modified version of Buses is implemented for the first time in this thesis.

The infrastructure required is a set of nodes running the Buses protocol and a semantically secure cryptosystem. The latter requirement prevents an eavesdropper from distinguishing between random data and an encrypted message within polynomial time. The sender requires the public keys or symmetric keys of the receiver and all intermediate nodes on the bus route from the sender to the receiver. Also, the sender is required to be able to buffer a bus. In addition, a CA or a trusted server is required to distribute the keys securely.

There are three types of bus anonymity schemes described in [4], each of which is optimal in a different sense. The optimal communication complexity version reduces the maximum number of messages that are sent simultaneously by the nodes in the network. The optimal buffer complexity version reduces the number of seats in a bus, thus reducing the size of the buffer required to store incoming and outgoing buses. The optimal time complexity version bounds the worst case time it takes two nodes to communicate anonymously. These basic schemes can be combined to trade off communication and time complexity.

In the optimal communication complexity version, the bus has n^2 seats, where n is the number of nodes in the system. Seat $s_{i,j}$ is reserved for communication from node p_i to node p_j. The jth column denotes all the seats reserved to communicate with node p_j. The bus traverses the network using an Euler tour of a spanning tree to define a ring. When node p_i receives the bus, it decrypts column i, retrieves any intelligible messages, and ignores the rest. Then, it replaces every seat $s_{i,j}$ in row i with either an encrypted message it wants to send to another node or detectable random data encrypted for p_j. The random data is encrypted because it prevents an attacker from determining how many seats a node is sending by timing how long it takes for the node to modify the bus. If a semantically secure cryptosystem is used, an adversary cannot distinguish the valid messages from the encrypted random data, and mutual anonymity is achieved. However, mutual anonymity is only provided from a third party's perspective, because the receiver can identify the sender by determining in which row i it received an intelligible message, and correlating this information to the sending node p_i by observing the bus as it traverses the network. Communication complexity is optimal because there is a single bus. It takes at most $2n-1$ time units for the bus to reach the sender, and at most an additional $2n-1$ time units for the bus to deliver the message to the receiver. The buffer complexity is $O(n^2)$ because there is a bus seat for every (source,destination) pair.

The optimal buffer complexity version reduces the size of the bus. Rather than having n^2 seats, there are s^2 seats, where s is an upper bound on the number of messages that are sent anonymously for each tour of the bus. When the bus reaches a node p_i, it decrypts *every* seat on the bus, replacing each seat with its decryption. Any intelligible messages are recorded and replaced with random data. In order for node p_i to send a message, it first prepares a layered encryption of the message. It encrypts the message with the destination node's key, encrypts the encrypted message with the destination's predecessor's key, and so on until it encrypts the multiply-encrypted message with its successor's key. This layered encrypted message is placed in a random seat on the bus and the bus is forwarded to p_i's successor. One layer of the layered encryption is removed at each hop in the bus tour of the network, until eventually the intended receiver removes the last layer of encryption and extracts the message. Mutual anonymity is provided from everyone's perspective, including the receiver, because the layered encryption only allows a receiver to identify the predecessor in the bus path, and not the original sender.

The optimal buffer complexity version of Buses has a bus of size $O(ns^2)$, time complexity $O(n)$, and communication complexity $O(1)$. It is assumed that $s \leq \sqrt{n}$, otherwise the buffer complexity is worse than the optimal communication complexity protocol. The buffer complexity is $O(ns^2)$ rather than $O(s^2)$ because encrypting with a semantically secure cryptosystem increases the size of the message with each layer of encryption. To prevent a size correlation of seats before and after the bus visits a node, the sender appends dummy blocks to the decryption. In the worst case, the receiver is $O(n)$ hops away in the Euler tour, so the buffer complexity is $O(n)$ times s^2, the size of the bus.

Since s is the upper bound of messages that are sent anonymously for each cycle of the bus, it is possible for a bus with buffer complexity $O(ns^2)$ to have a "birthday paradox" where two senders choose the same seat. The expected number of resends for a message due to the birthday paradox is less than 1 [4]. For reliable message delivery, the receiver sends an acknowledgment in the same seat. If the sender does not find an acknowledgment in the seat when the bus returns, the sender simply resends the message. If a sender has to send a message multiple times, it can double the size of a bus in order to decrease the probability that an existing message is accidentally overwritten. Similarly, if multiple acknowledgments are received, then the size of the bus can be halved.

To minimize the time required to send anonymous messages, the optimal time complexity variation increases the number of buses, with two buses traveling in each direction of each link in the network. If a message destined for node p_j arrives in seat $s_{i,j}$ at node p_k, it transfers to the bus on the link out of p_k that has the shortest path to p_j. One problem with this protocol is that network bandwidth is consumed in exchanging routing information for shortest paths. As in the optimal buffer complexity protocol, mutual anonymity is provided if layered encryption is used and the sender knows the shortest path to the receiver. The time complexity of this variation is O(diameter of the graph), the communication complexity is $O(2m)$, and the buffer complexity is $O(n^2)$, where m is the number of edges in the graph.

Lastly, a bus protocol that gives a trade-off between time and communication is possible. In fact, Theorem 4.1 of [4] states:

> For every x, where $1 \leq x \leq n$, there is an anonymous message delivery protocol with communication complexity $O(n/x)$, buffer complexity $O(n^2)$, ... the time complexity between two nodes is $O(min(dx, n))$, where d is the distance between the nodes in the spanning tree.

The idea is to divide the graph into clusters. An individual bus tours each cluster. Cluster size is directly related to communication complexity and inversely related to time complexity. In order to improve time complexity, each bus has to be synchronized to meet at common edge nodes of adjoining clusters. As in the optimal buffer complexity protocol, mutual anonymity is provided if layered encryption is used.

The Buses anonymous communication scheme can be extended to handle multicast and broadcast, unknown topology, and an active adversary. Multicast and broadcast can be implemented by having the node send an anonymous message to another random node to perform the multicast or broadcast on behalf of the anonymous sender. For an unknown topology, the bus can perform a random walk in expected time $O(nm)$ [1], where n is the number of nodes in the graph and m is the number of edges. An active adversary can add, delete, or modify messages and anonymous communication still succeeds. In order for the anonymous communication scheme to succeed against an active adversary controlling t nodes, a sender must send a message on $t + 1$ disjoint paths.

2.11 DC-Net and Xor-trees

DC-Net [8] was written in 1988 and provides mutual anonymity for message communication, based on the dining cryptographers problem [8]. DC-Net provides a very strong form of mutual anonymity at the cost of expensive broadcast communication. Due to this expensive broadcast communication, DC-Net is not used in practice.

The network consists of nodes and edges, and the nodes are completely connected. Pairs are formed by the two nodes incident to an edge. Each pair observes secret binary coin flips and calculates the total modulo 2 of all secret binary coin flips it observes (the parity). This parity is then broadcast to all other nodes. Since each binary coin flip is observed twice (since two nodes are incident to each edge), the total of the broadcast parities modulo 2 is zero. If a node wants to communicate with another node, it inverts the parity observed before broadcasting. This results in the total of the broadcast parities being odd for the receiver.

This idea can be extended by sharing symmetric keys; each participant has a symmetric key in common with every other participant. This allows for multiple coin flips, with the ith bit of the symmetric key representing the ith coin flip. However, only one node can communicate at a time, otherwise an even number of inversions results in an even parity. Even worse, if an odd number of nodes communicate, the inversion collisions are not detected. Another problem is that the results of each round must be broadcast to the entire network. Obviously, this scheme does not scale due to the broadcast communication and hence is not used in practice, even though it has been implemented [12].

An extension to DC-Net is Xor-trees [13]. This solution reduces the amount of communication by having nodes transfer the results of their secret coin flips (using each bit of the shared symmetric key as a coin flip) towards the root, with parent nodes Xoring the results from the sibling nodes. Once the results are propagated to the root, it broadcasts the result back down the tree. Public-key cryptography is used to distribute the shared symmetric keys, so Xor-trees requires a CA. However, this anonymous communication scheme has the same inherent problems of DC-Net. It only allows for one node to communicate at a time and has difficulty handling message collisions when more than one node communicates. Since Xor-trees is just a slightly more efficient version of DC-Net, it is not analyzed in Chapter 3.

2.12 P^5

Peer-to-Peer Personal Privacy Protocol (P^5) [37] was written in 2002 and is a theoretical idea that provides mutual anonymity for message exchanges. It is based upon a broadcast hierarchy in which each node in the hierarchy mixes its inputs and outputs and constantly generates a stream of signal and noise packets. P^5 allows individual users to decrease anonymity to improve efficiency; however, it is not possible for a user to increase anonymity after it has been decreased. The infrastructure required by P^5 is an Oracle and a logical broadcast tree. A sender requires the public key of the receiver (acquired via out-of-band communication— secure communication that does not use the anonymous communication protocol) and the channel of the receiver (see below). Thus, P^5 requires a CA as a trusted authority. P^5 has not been implemented yet.

A group of N users is partitioned into a logical broadcast tree, L, via their public keys, $K_0^+, K_1^+, \ldots, K_{N-1}^+$. Each node of L is referred to as a channel, and a subgroup of users could be listening to each channel. The notation (b/m) is used to represent the members of a subgroup where b is a bit-string and m is a bitmask (the length of the bit-string b). m is inversely related to communication and anonymity. For example, (01/2) represent all group members with the prefix 01 for the hash of their public key. A user A can join a group (b_a/m) where $m \geq 0$ and $b_a = H(K_A^+) \pmod{2^{L_d}}$. In this notation, H is a secure hash function, K_A^+ is the public key of A, and L_d is the depth of the tree.

When a message is sent to a channel, it is sent to all the members of the channel, their subtrees, and the path from the channel to the root (any channel with an ancestor or descendant relationship). A message is encrypted with the public key of the receiver. Any member of the channel that receives this message could be the receiver. A user that wishes to receive a message can advertise a channel as its message distribution point in the tree. The higher up in the tree the advertisement is (smaller m), the bigger the pool of potential users that could be the receiver. Any of the users in this channel pool is equally likely to be the receiver. To increase efficiency, a user can advertise itself lower down in the tree (larger m), but this decreases the pool of channels that could be the receiver, effectively decreasing the receiver's anonymity.

A centralized Oracle that is a trusted authority is assumed to exist. When the user joins a channel, the Oracle lets a user know how many members belong to that group. Depending on the user's desired efficiency, it can migrate down the tree at the cost of decreasing its anonymity.

To provide sender anonymity, users always generate traffic at a fixed rate destined for a randomly chosen channel. These packets can be a packet that was received from another node (either a signal packet that is a message packet or a noise packet which is a dummy packet), a signal packet that is generated locally, or a noise packet that is generated locally. The noise packets prevent a passive correlation attack. All packets are the same size to prevent size correlation. Some packets may be dropped, packets destined for higher nodes in L are dropped with higher probability.

2.13 Pipenet and Freedom

Another two anonymous communication schemes are Freedom [14] and Pipenet [30]. Pipenet is an extremely short and theoretical description of an anonymous communication scheme written some time around 2000 and is based upon mixes that use virtual link encryption. *Virtual link encryption* means that every single communication path between two mixes (every virtual link) is associated with a key (either a symmetric key or a public/private key pair). Before a message is sent across a link, it is encrypted with the link's symmetric/public key and after it traverses the link it is decrypted with the link's symmetric/private key. Pipenet is not used in practice. Due to the brevity of the description in [30], it is not possible to properly analyze it and it is omitted from the analysis in Chapter 3.

Freedom is a commercially available anonymous communication scheme similar to Onion routing written in 1999, but it only allows a static re-routing path length of three intermediate proxies. The company that implemented Freedom (Zeroknowledge) is not releasing any documents on Freedom, so it is not analyzed in Chapter 3. It is either a re-routing or mixing anonymous communication scheme. See [14] for an implementation of Freedom.

2.14 Summary

The application support, sender requirements, and infrastructure requirements of the anonymous communication schemes are summarized in Tables 2.1 and Table 2.2. Application support denotes the applications that can be used in conjunction with the anonymous communication scheme. Sender requirements denotes the information that the sender needs to participate in the anonymous communication scheme. It is implicit that the sender has the address of the receiver, except in Buses, DC-Net, and P^5 where it is not required. Infrastructure denotes what additional components are needed for users to communicate anonymously. PK denotes public key and SK denotes symmetric key. Not Applied means that it has not been applied to an actual application domain. E-mail (1-way) means that the recipient has no means to reply to the sender.

The anonymity techniques used and the kind of anonymity provided are summarized in Table 2.3. Weak Sender Anonymity denotes a re-routing technique with no mixing that relies on one intermediate proxy to provide sender anonymity. Weak Mix is a mix that uses a technique such as message delay instead of batching, which

Table 2.1: Anonymous Communication Schemes Summary: Application Support, Sender Requirements, Infrastructure Requirements

Scheme	**Application Support**	**Sender Requirements**	**Infrastructure Requirements**
Chaum's Mixes	- E-mail - E-voting	- Addresses of Mixes - PKs of Mixes - PK of Receiver	- Cascade of Mixes
Onion Routing	- VPN, rlogin - E-mail - Web Cash - Web Browsing	- Application Proxy - Onion Proxy Addr. - PK of Onion Proxy - PK of Receiver	- Onion Routing Network
Crowds	- Web Browsing	- Jondoe - Blender Account (Login & Password) - Blender Address	- Blender - Participating Crowd Members
Hordes	- Web Browsing	- Jondoe - Blender Account (IP Addr. of Initiator, PK of Blender, & PK of Sender) - Blender Address & Sender Private Key	- Blender - Multicast Support - Participating Horde Members
LPWA	- Web Browsing - Email	- Passphrase - Addr. of Janus Proxy	- LPWA Proxy
Anonymizer	- Web Browsing	- Address of Proxy	- Anonymizer Proxy
Type 0 Remailer	- E-mail (1-way)	- Address of Remailer	- Type 0 Remailer
Cypherpunk	- E-mail	- Addrs. of Remailers - PKs of Remailers	- Cascade of Cypherpunk Remailers
Mixmaster	- E-mail (1-way)	- Addresses of Mixes - PKs of Mixes	- Cascade of Mixmaster Mixes

Table 2.2: Anonymous Communication Schemes Summary (2): Application Support, Sender Requirements, Infrastructure Requirements

Scheme	**Application Support**	**Sender Requirements**	**Infrastructure Requirements**
Babel	- E-mail	- Addresses of Mixes - PKs of Mixes - Password (.forward Configuration) - First Hop's PGP Key (Proxy Configuration) - Receiver's PGP Key (Optional)	- Cascade of Babel Mixes
Buses	- Not Applied	- Bus Buffer Capacity - PK or SK of Receiver - PKs or SKs of Nodes on Route to Receiver	- Set of Nodes Running Buses Protocol - Semantic Security
DC-Net	- Not Applied	- Shared SKs with Neighbors	- Broadcast Support
P^5	- Not Applied	- PK of Receiver - Channel of Receiver	- Broadcast Support - Oracle - Logical Bcast. Tree

Table 2.3: Anonymous Communication Schemes Summary: Anonymity Technique, Anonymity Provided

Scheme	**Anonymity Technique**	**Anonymity Provided**
Chaum's Mixes	- Mixes	- Sender Anon. & Unlinkability
Onion Routing	- Mixes	- Sender Anon. & Unlinkability
Crowds	- Re-routing with Path Encryption	- Sender Anon. & Unlinkability
Hordes	- Re-routing with Link Encryption (Forward Direction) - Multicast (Backward Direction)	- Sender Anon. & Unlinkability
LPWA	- Janus Function - Re-routing	- Data Anon. - Weak Sender Anon.
Anonymizer	- Re-routing	- Weak Sender Anon.
Type 0 Remailer	- Re-routing	- Weak Sender Anon.
Cypherpunk	- Weak Mix	- Sender Anon. & Unlinkability
Mixmaster	- Mixes	- Sender Anon. & Unlinkability
Babel	- Mixes	- Sender or Mutual Anon. (RPI)
Buses	- Bus Metaphor	- Unlinkability or Mutual Anon.
DC-Net	- Shared Coin Flips	- Mutual Anon.
P^5	- Broadcast Hierarchies	- Mutual Anon.

does not obfuscate the inputs with the outputs of a mix under low load conditions.

The message size, mixing techniques, and path characteristics are summarized in Table 2.4. Hop-wise means that each node flips a biased coin. If the flip indicates to forward the message, the node chooses the next hop randomly.

If the anonymous communication scheme has replay protection, if the anonymous routing scheme uses routing tables to set up anonymous paths, and what type of encryption it uses is summarized in Table 2.5.

Table 2.4: Anonymous Communication Schemes Summary: Message Size, Obfuscation Techniques, Path Selector/Length

Scheme	**Message Size**	**Mixing Techniques**	**Path Selector/ Length**
Chaum's Mixes	- Uniform	- Batching (Lexicographic Reordering) - Constant Rate (Dummy Messages)	- Sender/Undefined
Onion Routing	- Uniform	- Batching (Pseudorandom Reordering)	- Onion Proxy/Length 5 (Onion I) - Hop-wise/Random (Onion II)
Crowds	- Not Uniform	- None	- Hop-wise/Random
Hordes	- Not Uniform	- None	- Hop-wise/Random (forward path)
LPWA	- Not Uniform	- None	- Fixed/Length 1
Anonymizer	- Not Uniform	- None	- Fixed/Length 1
Type 0 Remailer	- Not Uniform	- None	- Fixed/Length 1
Cypherpunk	- Not Uniform	- Delay - Maintain Message Pool (Reordering)	- Sender/Depends on Implementation
Mixmaster	- Uniform	- Delay - Reordering	- Sender/Length 20
Babel	- Uniform	- Interval Batching (Decoy Messages) - Probabilistic Deferment - Inter-Mix Detours - Fixed-set Random Order Path	- Sender/Length is Number of Mixes
Buses	- Uniform	- Always Decrypt Seats - Replace a Message Seat with Random Data	- Bus/Euler Tour - Bus/Random Walk
DC-Net	- Uniform (1 bit)	- Constant Rate	- N/A (Broadcast)
P^5	- Not Uniform	- Constant Rate	- Receiver/Dependent Upon Desired Anon.

Table 2.5: Anonymous Communication Schemes Summary: Replay Protection, Anonymous Path Routing Tables

Scheme	**Replay Protection**	**Anonymous Path Routing Tables**	**Encryption Type**
Chaum's Mixes	Yes	No	Layered Encryption
Onion Routing	Yes	Yes	Layered Encryption
Crowds	No	Yes	Path Encryption
Hordes	No	No	Link Encryption (Forward) None (Backward)
LPWA	No	Yes	None
Anonymizer	No	Yes	None
Type 0 Remailer	No	No	None
Cypherpunk	No	Yes	Layered Encryption
Mixmaster	Yes	No	Layered Encryption
Babel	Yes	No	Layered Encryption
Buses	No	N/A	Layered Encryption
DC-Net	N/A	N/A	None
P^5	No	N/A	Path Encryption

Chapter 3

Analysis of Vulnerabilities

In this chapter, all of the surveyed schemes in Chapter 2 are analyzed and the attacks that can be launched against each anonymous communication scheme successfully are identified. The threat models are defined in Section 3.1 and the attacks are described in Section 3.2. In Section 3.3, the attack vulnerabilities for each anonymous communication scheme are identified, the results are analyzed, and the features required to defend against all of the surveyed attacks for each anonymous communication technique are summarized.

3.1 Threat Models

In order to evaluate the effectiveness of the anonymous communication schemes, it is necessary to define the threat models. Each threat model represents a malicious attacker with a different amount of resources. The attacks can be classified into three threat models:

- An *eavesdropper* can see messages that traverse one or more links in the network. A *global eavesdropper* can see messages that traverse all the links in the network.
- A *passive adversary* can compromise one or more nodes in the system and can gather information about messages that pass through a compromised node,

such as who the predecessor and successor nodes are, and local information stored on the compromised node (e.g., routing tables, private keys).

- An *active adversary* has the same abilities as a passive adversary but can also insert or delete messages on any of the links of the compromised nodes.

Let *attacker* denote either an eavesdropper or an adversary.

3.2 Attacks

Most of the anonymous communication schemes prevent a compromised receiver from identifying the sender, however these are not the only attacks that can break an anonymous communication scheme. Denial of service attacks are not considered in this analysis, but attacks at the eavesdropper, passive adversary, and active adversary level are considered[1]. The definition of the attack, the characteristics that allow the attack, and the features that thwart the attack are defined below. The nomenclature and definitions of these attacks vary and are scattered throughout the literature — our sources are [4], [7], [9], [10], [17], [18], [35], [37], [38], and [41].

3.2.1 Eavesdropper Attacks

The **eavesdropper attacks** are:

- *Local eavesdropper before proxy* [9, 18, 35, 38, 41]. If the anonymous communication scheme sends messages in plaintext before the proxy/first mix/first re-

[1]In general, there are many possible sets of colluding nodes for each anonymous communication scheme. The analysis of all these sets is omitted unless defined explicitly in one of the attacks. The interested reader is referred to the original paper of the anonymous communication scheme, but is forewarned that this analysis probably does not exist for all possible sets.

router, then a local eavesdropper before the proxy can defeat sender anonymity. The attack is thwarted by encrypting the message before it is sent or by having the first proxy run as a local process on the sender's machine (this first proxy must then encrypt the message).

- *Local eavesdropper after proxy* [9, 18]. If the anonymous communication scheme sends messages in plaintext from the last proxy/mix/re-router to the receiver, then a local eavesdropper after the proxy can defeat receiver anonymity. The attack is countered by encrypting the message with a key that only the receiver knows.

- *Size correlation* [4, 7, 9, 10, 18, 37, 41]. If messages are different sizes, then a global eavesdropper can trace the message from sender to receiver using size correlation. This defeats all types of anonymity. The attack is defeated by using uniform message size. This attack is not applicable to broadcast techniques that send messages directly to the receiver because there is no message size to trace as it is routed through the network.

- *Time correlation* [7, 9, 10, 18, 37, 38, 41]. If no mixing techniques are enforced, a global eavesdropper can trace a packet from sender to receiver. This defeats all types of anonymity. The attack is stopped by using mixing techniques. This attack is not applicable to broadcast techniques that send messages directly to the receiver via broadcast because there are no input/output times at intermediate proxies to correlate. Also, the attack does not apply to Buses because time correlations only reveal bus route(s), not message patterns.

- *Low load attack* [4, 10, 18] (also similar to *Volume attack* [41]). If the anonymous communication scheme does not enforce a minimum level of traffic via dummy messages, then under low load conditions all anonymity types are statistically reduced and in the worst case defeated. A global eavesdropper can keep track of the number of messages that pass through each individual node. Then, the message counts between each node can be correlated to determine communication paths statistically. This attack is foiled by a minimum level of traffic enforced by dummy/decoy messages. This attack does not apply to Buses because a bus always travels its route and each node always decrypts and/or replaces a set of seats. Also, the attack does not apply to broadcast schemes that broadcast the message directly to the receiver group.

- *Marker attack* [9, 10, 18, 38, 41]. If an anonymous re-routing communication scheme only encrypts an incoming message at the first hop and decrypts an outgoing message at the last hop or does not encrypt a message at all, a common prefix of a message can be traced from source to destination. This defeats sender and receiver anonymity. The attack is evaded by layered encryption or link encryption. This attack does not apply to broadcast techniques or multicast techniques. For broadcast techniques, messages are sent directly to the receiver via broadcast and the attacker only identifies that every node that receives the broadcast is equally likely to be the receiver. Similarly for multicast every node in the multicast group is equally likely to be the sender to whom the receiver is replying.

- *Intersection attack* [37]. Suppose that an eavesdropper knows that a sender belongs to two different sets U and V. Then, the adversary knows that the sender belongs to the intersection of these two sets. This reduces all types of anonymity. Out of all the anonymous communication schemes surveyed, this attack is only applicable to P^5 where a user migrates down the logical broadcast hierarchy binary tree L. The attack is thwarted by not migrating down L.

3.2.2 Passive Adversary Attacks

The **passive adversary attacks** are:

- *Full compromised path* [38]. If an attacker can compromise all the nodes in a path, it can determine the source and destination nodes. This defeats all types of anonymity. The attack is countered by constant rate traffic with dummy messages and layered encryption, or link encryption. Constant rate traffic with link encryption or layered encryption prevents an adversary from knowing if it has compromised the full re-routing path, or just a sub-set of it. This attack does not apply to Buses. Constant rate traffic is not needed because buses always travel their routes and semantic security prevents a passive adversary from determining in polynomial time if a seat was replaced with random data or an encrypted message. This attack does not apply to broadcast or multicast schemes that send the messages directly to the receiver.

- *Timing attack* [35]. When Web browsing, a user requests Web pages. If a requested Web page has embedded objects, the browser automatically requests them after receiving the Web page. If a passive adversary notices significantly short times between the embedded object requests, it can with high probability identify the predecessor as the sender and defeat sender anonymity. The attack is defeated by constant rate traffic with dummy messages and batching or by the last hop in the anonymous path checking if there are embedded objects and requesting them on behalf of the sender that initiates the request. In the latter case, the sender's web browser is changed not to request embedded objects, but to wait for them. This attack does not apply to Buses because embedded objects are sent on the next available bus, not immediately. Also, this attack does not apply to broadcast techniques because there are no intermediate nodes between sender and receiver that can identify the sender as a predecessor.

- *Passive traceback attack* [17, 38]. If a passive adversary can start at a receiver and then corrupt each routing table on the backward path of the message, it defeats sender anonymity. However, this requires these paths to be stored in a routing table. There is currently extensive work in this field because of the need for network administrators to identify intruders. The attack is stopped by anonymous message paths not being stored in routing tables. Note that this attack is passive only if the adversary's actions to corrupt routing tables are not noticeable (i.e., it does not create a noticeable event). Otherwise, the attack is considered an active adversary attack.

3.2.3 Active Adversary Attacks

The **active adversary attacks** are:

- *Replay attack* [7, 9, 10, 18, 41]. An active adversary can repeat (multiple times) a message that it has observed on the network. It then sees multiple identical messages pass through the network and can correlate them to the message it replayed. This defeats receiver anonymity for re-routing/mixes and decreases sender anonymity. This attack also defeats receiver anonymity for Buses. When a node in Buses receives a valid message, it replaces the message with random data. Hence a replay results in a different output Bus for a node if it received a valid message. If the node did not receive a valid message it results in an identical output Bus. The attack is foiled by preventing replays via timestamps, et cetera. The attack does not apply to broadcast schemes that broadcast messages directly to the receiver because there are no paths to trace for identical messages.

- *Spam attack* [9, 10, 18]. If batching and replay protection are used, an active adversary can spam a proxy with multiple messages that it creates itself. Since the active adversary knows what the outgoing messages of the proxy look like for the messages the adversary creates, it can isolate and identify a sender's message. This attack defeats receiver anonymity and decreases sender anonymity. Note that a spam attack for a non-batching technique is unnecessary and a replay attack suffices. The attack is evaded by using message delay within the message itself, probabilistic deferment, or inter-mix detours.

- *Mob attack* [35, 37]. An active adversary can join the anonymous communication scheme and produce a large amount of traffic, reducing the protocol's performance. This attack is only applicable to P^5 because it can cause a sender to migrate down the tree L, reducing sender anonymity via an intersection attack. The attack is thwarted by preventing migrations down L.

- *Filter attack* [10] (similar to a *trickle attack* [18]). If delay is used, an adversary can delete all messages intended for a mix, except one. When the next message is sent, it can only be the message that was not deleted. The attack is countered by using batching.

3.3 Analysis

3.3.1 Attack Susceptibilities

The attacks that each anonymous communication scheme are susceptible to at the eavesdropper, passive adversary, and active adversary level are summarized in Tables 3.1 and 3.2. If an anonymous communication scheme is not susceptible to any of the attacks presented in this thesis for a particular level of attacker, it is labeled "secure". Otherwise, the attacks that the anonymous communication scheme cannot defend against are indicated. If the attack has been identified before, reference(s) to where it was identified are given in Table 3.3. If the attack is followed by a *, then the attack was previously unknown to the best of the author's knowledge. These new attacks were identified by cross-referencing the characteristics that defend against each attack in Section 3.2 against the anonymity schemes' characteristics in Tables 2.1, 2.2, 2.3, 2.4, and 2.5. Note that if a scheme is susceptible to a replay attack, then it is implicitly susceptible to a spam attack (this implication is not recorded in Tables 3.1 and 3.2).

Table 3.1: Anonymous Communication Schemes Attack Vulnerabilities

Scheme	**Eavesdropper**	**Passive Adv.**	**Active Adv.**
Chaum's Mixes	- Secure	- Secure	- Spam attk.*
Onion Routing	- Low load attk.*	- Full comp. path - Timing attk.* - Pass. tb. attk.	- Spam attk.* - Marker attk. (onion routers)
Crowds	- Size corr. attk.* - Time corr. attk. - Low load attk.* - Marker attk. (path encryption) - Loc. evdpr. aft. pxy. *	- Full comp. path* - Pass. tb. attk.	- Replay attk.*
Hordes	- Size corr. attk. (fwd)* - Time corr. attk. (fwd) - Low load attk. (fwd)* - Loc. evdpr. aft. pxy. *	- Full comp. path - Timing attk.*	- Replay Attk.*
LPWA	- Loc. evdpr. bf. pxy. - Size corr. attk. - Time corr. attk. - Low load attk. - Loc. evdpr. aft. pxy. * - Marker attk.*	- Full comp. path* - Timing attk.* - Pass. tb. attk.*	- Replay attk.*
Anonymizer	- Loc. evdpr. bf. pxy. - Size corr. attk. - Time corr. attk. - Low load attk. - Loc. evdpr. aft. pxy. * - Marker attk.	- Full comp. path* - Timing attk.* - Pass. tb. attk.*	- Replay attk.*
Type 0 Remailer	- Loc. evdpr. bf. pxy. - Size corr. attk.* - Time corr. attk.* - Low load attk.* - Loc. evdpr. aft. pxy. * - Marker attk.	- Full comp. path*	- Replay attk.*
Cypherpunk	- Size corr. attk. - Low load attk. - Loc. evdpr. bf./aft. pxy. *	- Full comp. path* - Pass. tb. attk.*	- Filter attk. - Replay attk.
Mixmaster	- Low load attk. - Loc. evdpr. bf./aft. pxy. *	- Full comp. path*	- Filter attk.* - Spam attk.*

Table 3.2: Anonymous Communication Schemes Attack Vulnerabilities (2).

Scheme	**Eavesdropper**	**Passive Adv.**	**Active Adv.**
Babel	- Loc. evdpr. aft. pxy. * (.forward configuration)	- Full comp. path*	- Secure
Buses	- Secure	- Secure	- Replay Attk.*
DC-Net	- Secure	- Secure	- Secure
P^5	- Intersection Attk.	- Secure	- Mob attk.

3.3.2 Required Defensive Attributes

Upon examining Column 2 of Tables 3.1 and 3.2, it is clear that certain characteristics are required for anonymous communication schemes to defend against an eavesdropper. Uniform message size is required to prevent size correlation attacks. To prevent a local eavesdropper before/after a proxy, re-routing and mixing schemes should encrypt the message before the first proxy/mix and have the message encrypted so that only the receiver can decrypt the final message. For mixing techniques, constant rate traffic and batching or probabilistic deferment should be used to prevent time correlation and low load attacks. A path key for path encryption is not a good technique to improve efficiency because a global eavesdropper can perform a marker attack.

Note that a local eavesdropper before/after a proxy can easily be thwarted by making an encrypted connection to the receiver that does not authenticate the sender. However, most encrypted connections provide the functionality for the receiver to authenticate the sender, such as PGP. Clearly, this authentication defeats sender anonymity.

Thus, a secure anonymous communication scheme at the eavesdropper level should either use Buses, DC-Net, or a mixing scheme that uses uniform message size, message encryption before the first proxy so that only the receiver can decrypt the final message, constant rate traffic and batching or probabilistic deferment. The only mix scheme discussed in this thesis that satisfies these characteristics is Chaum's mixes.

Table 3.3: Anonymous Communication Schemes Attack Vulnerabilities: Sources from Literature

Scheme	**Sources of Attacks in the Literature**
Onion Routing	Full comp. path [38] - Pass. tb. attack [38] - Marker attack [41]
Crowds	- Time corr. attack [37, 38] - Pass. tb. attack [38] - Marker attack [38]
Hordes	- Full comp. path [38] - Time corr. attk. (fwd) [37]
LPWA	- Loc. evdpr. bf. pxy. [9, 41] - Size corr. attack [9] - Time corr. attack [9] - Low load attack [41]
Anonymizer	- Loc. evdpr. bf. pxy. [9, 41] - Size corr. attack [9, 41] - Time corr. attack [9, 41] - Low load attack [41] - Marker attack [41]
Type 0	- Loc. evdpr. bf. pxy. [9] - Marker attack [41, 10]
Cypherpunk	- Filter attk. [10] - Size corr. attk. [9, 10, 15] - Low load attack [10] - Replay attack [9, 10, 15]
Mixmaster	- Low load attack [10]
P^5	- Intersection Attack [37] - Mob attack [37]

Re-routing schemes are not secure against global eavesdroppers. Most of the re-routing schemes say that it is unlikely that an eavesdropper could span multiple administrative zones. However, assuming the absence of a global eavesdropper opens the door to attacks. For example, an Ethernet network uses a broadcast medium so any node on that Ethernet network can act as a global eavesdropper for that network (this node could be a valid user corrupted by malicious software or a valid but malicious user). Thus, an attacker can corrupt the multiple Ethernet networks spanned by an anonymous protocol, and become a global eavesdropper. Hence, re-routing schemes should employ some obfuscation techniques, otherwise they are susceptible to a global eavesdropper defeating anonymity because of time correlations.

Column 3 in Tables 3.1 and 3.2 shows that certain characteristics are required for anonymous communication schemes to defend against a passive adversary. To prevent a full compromised path attack, a re-routing or mixing scheme should use constant rate traffic with dummy messages and single encryption, layered encryption, or link encryption. To prevent a passive traceback attack, a re-routing/mix scheme should not use routing tables. If the application support includes web-browsing, a timing attack must be defended against. A timing attack is countered by constant rate traffic with dummy messages and batching. Alternatively, a timing attack is countered by the last hop in the anonymous path checking if there are embedded objects and requesting them on behalf of the sender that initiates the request.

A secure anonymous communication scheme at the passive adversary level should either use Buses, DC-Net, P^5, or a mixing scheme that does not use an anonymous routing table, uses constant rate traffic with dummy messages and encryption (single, layered, or link encryption), and uses batching if the application support includes

Web browsing. An alternative for batching is the last hop in the anonymous path checking if there are embedded objects and requesting them on behalf of the sender that initiates the request. The only mix scheme discussed in this thesis that satisfies these characteristics is Chaum's mixes.

Finally, Column 4 in Tables 3.1 and 3.2 shows that certain characteristics are required for anonymous communication schemes to defend against an active adversary. Mixes and re-routing schemes should use message delay within the message itself, probabilistic deferment, and inter-mix detours to prevent a spam attack. Re-routing schemes, mixes, and buses should use replay protection to prevent a replay attack. If delay is only used, re-routing schemes/mixes are susceptible to a filter attack. A filter attack is defeated by using batching.

A secure anonymous communication scheme at the active adversary level should either use Buses with replay protection, a broadcast scheme based upon DC-Net, or a mixing scheme that uses message delay within the message itself, probabilistic deferment, inter-mix detours, and replay protection. The only mix scheme discussed in this thesis that satisfies these characteristics is Babel.

Table 2.4 identifies additional characteristics required of anonymous communication schemes (based upon re-routing and mixing) to be secure. Path selection should be done by the sender. Otherwise, if the first node in the re-routing path is a compromised node, it can then choose the next hop from the set of compromised nodes. In the worst case, the entire path consists of compromised nodes and anonymity is defeated. Also, re-routing schemes that have a noticeable send event (such as in Crowds during re-routing path creation) should have a fixed path length. Otherwise, the analysis in [35] shows that sender anonymity is reduced. If a re-routing scheme does not have a noticeable send event (such as Chaum's mixes), the analysis in [17] shows that dynamic paths are stronger. However, this analysis does not address the effects on anonymity degree when multiple messages are sent by the same sender.

In summary, a secure anonymous communication scheme at the eavesdropper, passive adversary, and active adversary level should either use Buses with replay protection, DC-Net, or mixes with:

- uniform message size,
- encryption before the first proxy,
- encrypting the message so that only the receiver can decrypt the message,
- constant rate traffic,
- batching,
- probabilistic deferment,
- no anonymous routing table,
- constant rate traffic with dummy messages,
- message delay,
- probabilistic deferment,
- inter-mix detours,
- replay protection,
- path selection by the sender, and

- dynamic path length (not noticeable send event) or fixed path length (noticeable send event).

None of the mix schemes discussed in this thesis satisfies these characteristics. DC-Net is the only secure scheme but it is not scalable, making it impractical. Mixes with all the required features are difficult to implement correctly, and may lead to excessive communication (i.e., constant rate traffic with dummy messages). However, Buses with replay protection defends against all of the attacks and do not depend on the statistical properties of the traffic (i.e., do not require constant rate traffic with dummy messages).

3.4 Summary

There are numerous techniques that are used to provide connection anonymity. Re-routing techniques can improve efficiency at the cost of decreasing anonymity by allowing more attacks. However, if strong anonymity guarantees are required in the face of all threat models, a cascade of mixes is better. Thus, there is a direct correlation between anonymity and overhead for re-routing and mixes anonymous communication schemes.

However, this is not the case for schemes based on techniques other than mixes or re-routing. Buses provide an excellent level of anonymity at a manageable overhead and are susceptible only to replay attacks, which can be easily thwarted. Schemes such as DC-Net, Xor-trees, and P^5 are interesting from a theoretical point of view, but the first two are not practical since only one sender can communicate at a time and they are not scalable. The latter is susceptible to a mob attack launched by one adversary with minimal resources.

Chapter 4

A Practical Buses Protocol

This chapter contains an overview of the Practical Buses protocol, a modified version of the original Buses protocol [4]. In Section 4.1, the selection of Buses as a promising and practical protocol is provided. In Section 4.2, three new techniques to make the Practical Buses protocol support a dynamic topology while preserving mutual anonymity are introduced. Sections 4.3, 4.4, and 4.5 contain a description of the functionality added to the Buses protocol to enhance security and anonymity, improve fault tolerance, and improve performance. Lastly, an overview of the Practical Buses protocol is given in Section 4.6.

4.1 Justification for Buses

Our goals for a practical anonymous scheme are for it to be secure against all known attacks, provide scalability, provide mutual anonymity, and support a dynamic topology. The latter requirement is required to support dynamic membership, to recover from DoS attacks, and to support networks with a dynamic topology such as wireless networks. The assessment of each of these criteria identifies Buses as the best candidate. In addition, the protocol should be designed to sit between the transport layer and application layer in the protocol stack to provide application independence transparently to the user.

From Chapter 3, the best candidate for each anonymity technique that resists the most attacks is:

- mixes— Chaum's Mixes
- buses— Buses protocol (optimal buffer complexity)
- broadcasting— DC-Net
- re-routing— eliminated as a contender because they improve the performance of mixes at the cost of being vulnerable to more attacks.

DC-Net is already secure against all surveyed attacks, but Buses and mixes can be made secure against all surveyed attacks as well. Chaum's mixes can have inter-mix detours and delay within the message added to protect against a spam attack. Buses could include replay protection to protect against a replay attack.

Chaum's mixes and DC-Net are not scalable but Buses is scalable. Chaum's Mixes is not scalable because it requires constant traffic and continuous processing by all nodes in the network. DC-Net is not scalable because it requires broadcast communication, continuous processing by all nodes in the network, and only one sender can send a message at a time. Buses is scalable because it does not require constant traffic, continuous processing, or broadcast communication. In addition, the Buses network topology can be clustered. Nodes are only busy once per bus tour and a bus traverses only one link in the network at a time.

Buses and DC-Net provide mutual anonymity but Chaum's Mixes does not. Semantically secure layered encryption is the key to providing mutual anonymity in Buses. The dining cryptographers problem is the basis of providing mutual anonymity in DC-Net. Chaum's Mixes cannot provide mutual anonymity because the sender is required to know the receiver's address, and this can then be correlated to the identity of the user at that address.

Buses and DC-Net can support a dynamic topology, but mixes cannot. Buses can support a dynamic topology by having the bus perform a random walk and improving the protocol as described in Section 4.2 to ensure mutual anonymity is preserved. DC-Net can support a dynamic topology because messages are not re-routed, however it remains unresolved how a new node would join the network and exchange symmetric keys with the other participants without identifying itself as a sender. Mixes require a static set of mixes to re-route the message; otherwise messages will be lost before the layered encrypted message is delivered to the receiver because each hop in the re-routing path is required to remove a layer of encryption and forward the embedded encrypted message.

Hence, Buses is the only anonymous communication scheme that can meet all of our design goals. It is secure against all known attacks and provides scalability, mutual anonymity, and support for a dynamic topology.

4.2 Dynamic Topology Support and Mutual Anonymity

None of the Buses variations proposed in [4] nor any combination can support a dynamic topology and maintain mutual anonymity. The only Buses variation that could potentially do so is the optimal buffer complexity variation with layered encryption. All the other variations presented in [4] rely on reserved seats; these do not provide mutual anonymity because a receiver can identify the sender of the message by keeping track of who places messages in which seats. Even if the sender sends the message indirectly through a chain of nodes, the receiver can trace the message back to its original sender in a low traffic network. The optimized buffer Buses protocol requires semantically secure layered encryption on the reverse path of sender to receiver, but in a dynamic topology it is not possible to know the reverse path of sender to receiver because the bus route is dynamically changing. To resolve this contradiction, new techniques presented in Sections 4.2.1 to 4.2.3 are invented and comprise the core of the Practical Buses Protocol.

4.2.1 Nested Encryption with Indirection

The layered encryption approach is modified to handle dynamic network topologies with a novel solution denoted *nested encryption with indirection*. The problem with using layered encryption in a dynamic network topology is that the path from sender to receiver is unknown. Nested encryption with indirection chooses at random an indirection path of random bounded length to the receiver. Then, nested encryption is performed on the reversed indirection path. Of course, the next node to receive the bus may not be the first node in the indirection path. Thus, when the next node to receive the bus decrypts all the seats, any valid messages are forwarded by copying the decrypted inner layer to one of a set of seats designated as belonging to that node, and all other seats on the bus are not modified.

One method of determining whether a decrypted message is valid is to add fixed redundancy to each message that the node will detect upon decryption if the message was indeed intended for it. When a node decrypts a message intended for it, a flag is used to indicate whether the data obtained is a message intended for the node or a nested encrypted message that the node should forward to the next node in the indirection path. If the node is the final recipient, it extracts the message and replaces it with random data. Otherwise, it treats the nested encrypted message opaquely and simply forwards it.

In this scheme, an adversary can potentially glean information about communication patterns by observing when and how a node modifies messages intended for it on the bus. This problem of identifying a sender or receiver is resolved by the next two modifications to the protocol, respectively.

4.2.2 p-threshold Replacement Back-off Scheme

In order to preserve sender anonymity, the framework required is that each node owns k seats on the bus which it uses to insert messages to send or forward. The owner of a seat is identified by a field containing the owner's public key. In addition to providing the framework for sender anonymity, owned seats prevent a node from overwriting a valid message placed by another node. This prevents collisions and improves performance by preventing a waste of resources.

To prevent an adversary from defeating sender anonymity by observing when and how a node modifies its seats, the actual message insertions are disguised. Every time a node receives a bus, it replaces exactly pk seats chosen at random from the k seats that it owns, where p is a constant fraction of seats to be replaced. Even if the node has no messages to send, it still replaces pk seats with random data. For this scheme to work, an adversary must not be able to distinguish between random data and the encryption of a meaningful message; thus a semantically secure cryptosystem is required.

In addition to preserving sender anonymity, the p-threshold replacement scheme must also address problems related to network topology. Because the bus takes a random walk through the network, a node cannot assume that the next hop in the indirection path of a message would receive the bus before the bus returns to the node. Thus, it is possible that a node sends a message or forwards a nested encrypted message in one of its seats and then deletes it prematurely upon receiving the bus, because the node for which it was intended has not yet received the bus. In the case of forwarding a nested encrypted message, this situation is prevented by incorporating a 128-bit randomly-generated seat tag into each seat. Each node keeps a record of the tags of all the seats it has forwarded but have not yet been received, and does not replace any seats that match tags in this list. As a result, p must be restricted so that all k seats are not changed as part of the replacement strategy.

In the case of resending a message, yet more precautions are necessary because multiple resends of a message on a low traffic network can weaken sender anonymity. If an adversary sees the exact same seat after the bus passes through the node owning that seat m times, then with probability $1 - p^m$ the seat contains a meaningful message being resent. The adversary should not be able to determine if a message is being sent, even in a low traffic network, nor reduce the possible set of senders. To solve this problem, after the number of resends exceeds a threshold t, a new indirection path for the message is selected and the message is re-encrypted. Random 128-bit salt values are added to the beginning of each indirection layer before encryption to ensure that the new nested encrypted message differs from the previous one.

The threshold t should be chosen as small as possible to limit the information an adversary can obtain about who is sending messages. However, if t is too small, then resources are wasted by the sender sending multiple disguised messages. Hence, a back-off on the threshold t is introduced so that the threshold corresponding to a given message is increased linearly each time it is re-encrypted.

4.2.3 Randomly Delayed Seat Deletion

Finally, a mechanism is required to preserve receiver anonymity while using nested encryption with indirection and p-threshold replacement back-off scheme. In addition, a node needs to know when the next hop in the indirection path has received a seat containing a valid message, so that it can stop forwarding or resending the nested encrypted message according to the p-threshold replacement back-off scheme. A node can recognize that the valid seat has been received when it has been deleted by another node. However, in a low traffic network the receiver of a message can be identified because it will delete the seat, and if no other messages are currently in transit, there will be a period of inactivity on the network.

In order to solve this problem, a node only deletes a valid seat in our modified protocol after it receives the bus a certain, randomly-chosen number of times. Then, it is possible that any node in the indirection path could also be the last node to delete a seat before the period of network inactivity. If a stronger guarantee is needed, the node could send an anonymous message to another node requesting that it delete the seat. When the node sees that the seat has been deleted, it can stop sending the deletion request — the deletion implicitly acknowledges the request.

4.3 Security and Anonymity Enhancing Modifications

To ensure anonymity is preserved, replay protection is incorporated in Section 4.3.1. In Section 4.3.2, we describe how anonymous acknowledgments are added to the protocol. Anonymous acknowledgments ensure that a message is delivered to the receiver. In Section 4.3.3, we explain how digital signatures are added to the protocol. Digital signatures ensure that an adversary's tampering of a seat is detected.

4.3.1 Replay Protection

The original Buses protocol is susceptible to a replay attack. The ability to replay messages can compromise anonymity because an adversary can replay a valid message and attempt to determine the intended recipient by observing the seat transformations as it passes through the indirection path to the receiver. The receiver is identified because its transformation will be the first difference, because it replaces the valid message with a random data seat. The modifications presented so far to the Practical Buses protocol also allow for a replay attack, even though a received message is not replaced with random data. A replay attack in the Practical Buses protocol can reduce the receiver's anonymity by observing the set of potential receivers (the nodes that deleted seats) in a low traffic network and taking the intersection of these observed sets. Thus, replay protection is still required.

To prevent replays, when a node receives a message, it records the decrypted inner layer as well as the value of a 128-bit salt field. The salt field is a random 128-bit number included in each layer of the nested encryption. The combination of the salt and message body uniquely identify a nested encrypted message. To prevent a node from indefinitely collecting replay protection entries, they are only kept until a replay protection time-out elapses. However, we now must prevent an adversary from resending a seat after the time-out has expired. To accomplish this, a timestamp field is appended to the seat and a signature is constructed (see Section 4.3.3 for details on signatures) on the entire seat including the timestamp to prevent tampering. Received seats with a timestamp predating the replay protection time-out are immediately deleted and ignored.

Replay protection is also required to prevent a message being received multiple times by a receiver due to legitimate resends (see Section 4.4.1 for details on resends) by the sender. Messages received by a receiver utilize the same replay protection except the message tag field, a random 128-bit number in the inner core of the nested encrypted message, is recorded instead of the salt field.

4.3.2 Anonymous Acknowledgments

The original Buses protocol uses acknowledgments for handling seat collisions, but does not indicate how to send an acknowledgment to an anonymous sender, nor how to tolerate network faults. For example, a network fault could result in a seat containing a nested encrypted message being corrupted in transmission due to a network error. As a result, the node for which it was intended would not recognize it as valid after decryption, and the message would not reach its final destination.

Acknowledgments are used in our modified protocol to address these problems. A sender incorporates a message tag in the innermost layer of encryption so it can uniquely identify an acknowledgment to a valid message that may be sent multiple times, such as 'yes' in a chat program. After receiving a message, the receiver sends back to the sender an anonymous acknowledgment message that includes the sender's 128-bit message tag.

The actual acknowledgment message (the innermost layer) must be encrypted using a different public key than the one used in the sender's seat owner field. Otherwise, an adversary could correlate the public key to the node that modifies the seats (with the same public key in the owner field) and defeat sender anonymity. A separate *anonymous public key* must be used if a sender wants to remain anonymous from the perspective of the receiver. The sender generates a second public/private key pair, and this new anonymous public key is inserted into the anonymous public key field in the inner core of the nested message. The receiver of the message encrypts the acknowledgment message using the supplied anonymous public key. While this scheme does solve the problem of ensuring sender anonymity, it has the unfortunate consequence that each received seat must be decrypted with both the private key corresponding to the public key in the seat owner field and the private key corresponding to the anonymous public key, in effect doubling the computations that a node has to perform upon receiving the bus.

The deletion of an acknowledgment message must be randomly delayed as described previously. Otherwise, the receiver could identify the sender in a low traffic network, based on the last node to delete a seat.

4.3.3 Signed Bus Seats

In order to detect unauthorized modifications to the contents of a seat, malicious or otherwise, the entire contents of each seat are digitally signed with a signature scheme with appendix such as RSA-SSA (RSA Signature Scheme with Appendix) [22].

4.4 Fault Tolerance

The Practical Buses protocol should be able to handle lost or corrupted seats/buses, whether the losses or corruptions are intentional or not. To handle lost or corrupted seats, resends are incorporated in Section 4.4.1. To handle lost or corrupted buses, Section 4.4.3 contains a description of dynamic bus creation and is based upon the initial bus creation in Section 4.4.2.

4.4.1 Resends and Acknowledgments

The original Buses protocol did not allow for messages to be resent automatically by the protocol. The Practical Buses protocol provides reliable message delivery. If a sender does not receive an acknowledgment within a time-out period, it resends the message via a new indirection path with nested encryption. After a maximum resend time, the protocol delivers an error to the application that makes the user aware that the message was not received by the receiver. When a node receives an acknowledgment, it can stop resending the message. This avoids wasting resources if the maximum resend time is not reached before the ACK time-out.

4.4.2 Initial Bus Creation

When a node joins the Buses network, it broadcasts an inquiry to a designated Buses port including the joining node's IP and public key. A node receiving this inquiry takes one of two possible actions. If a bus already exists, then it sends the 2-tuple reply {public key, IP} to the sender. If a bus does not exist, in addition to sending the 2-tuple reply the node initiates a leadership election [25]; a broadcast-convergecast of public keys. The leader is the node who has the largest public key and once elected creates a bus and forwards it. In order to determine if a bus exists, each node includes in its state a boolean busExists, initialized to false. The boolean busExists is changed to true upon the receipt of a bus, and changed to false if a bus time-out lapses and the node has not received a bus.

4.4.3 Bus Loss Tolerance

Bus loss tolerance is handled by the boolean busExists and the bus time-out. If the bus time-out lapses while a node is waiting for a bus, the busExists is set to false and a leader election is invoked. If any other node has the busExists boolean still true, it uses the sentinel value of $2^{\text{public key size}}$ instead of its public key in the convergecast, and this prevents the node possessing the largest key from being elected and creating a bus. In the event the bus was actually lost, the node that received the bus last before the bus was lost initiates the final leadership election which results in a leader being elected to create a new bus. No other node could still have busExists set to true, because all other nodes would have received the bus after the node that received the bus last, and all their busExists time-out would have lapsed setting busExists to false.

4.5 Performance Modifications

To improve performance, the Practical Buses protocol includes hybrid encryption, dynamic bus seats, expired bus seats, and reduced seat decryptions. Hybrid decryption is significantly faster than public key decryption, and this is important because every node that receives the bus must do a decryption for every single seat on the bus. Dynamic bus seats allows a node to specify how many seats it owns on the bus. Expired bus seats allow a node to delete a seat that was left by another node that departed and did not delete its seats. Reduced seat decryptions reduces the number of seat decryptions required when processing a received bus.

4.5.1 Hybrid Encryption

The Buses protocol requires a semantically secure cryptosystem so that an adversary cannot distinguish seats containing valid nested encrypted messages from seats containing random data in polynomial time. In practice, semantically secure cryptosystems, and indeed public-key cryptosystems in general, are too slow for bulk data encryption. Thus, our protocol uses a hybrid encryption scheme. A layer of the indirected encrypted message in our protocol consists of a session key encrypted using an asymmetric cipher with Optimal Asymmetric Encryption Padding (OAEP), such as the 1024-bit RSA-OAEP [21] public-key encryption scheme, followed by the inner layer encrypted using the session key with a symmetric cipher, such as 128-bit AES [27]. The salt field at the beginning of each encryption layer doubles as the symmetric key. Although we cannot prove semantic security for this scheme, it is unknown how to distinguish symmetric ciphertexts such as AES encryptions from random data in polynomial time, largely because ciphertexts produced by the AES block cipher are statistically random [39]. For clarity, even though the hybrid encryption scheme is modular, for the rest of the thesis assume that the hybrid encryption scheme consists of RSA-OAEP/CBC-AES. 1024-bit keys are used for RSA-OAEP and 128-bit keys are used for AES.

Using RSA-OAEP to encrypt session keys along with the messages themselves has two important advantages. First, symmetric keys need not be exchanged prior to communication as the required key is sent along with the message. However, the main advantage is that the RSA-OAEP cryptosystem is a realization of a plaintext-aware cryptosystem. Although plaintext-awareness cannot be proved for RSA-OAEP, it is believed to be computationally infeasible for an adversary to forge a valid ciphertext [33]. Thus, in order to determine whether a bus seat contains a valid message, a node only needs to decrypt the session key rather than the entire message. If the message contains the prescribed redundancy, the node extracts the session key and decrypts the rest of the message.

4.5.2 Dynamic Bus Seats

The original Buses protocol did not account for users with different resource demands. The Practical Buses protocol allows a user to specify how many bus seats they own, up to a predefined limit. With more seats, the bookkeeping and seat replacement duties increase, but the user can send more messages per bus tour.

4.5.3 Expired Bus Seats

The Practical Buses protocol improves upon the original Buses protocol by checking for seat timestamps that have expired. This event can occur when a node unexpectedly drops out of the protocol and does not delete its seats. To keep the buffer size as small as possible, if a seat has expired it is deleted by whatever node that currently has the bus. If the node that owns the deleted seat is still active, when it receives the bus it inserts new seats to replace the lost seats.

4.5.4 Reduced Seat Decryptions

The number of seat decryptions required when a node receives a bus can be reduced. Each time a node receives a bus, it sees approximately $(1 - p)\%$ of all the seats repeated as a result of the p-threshold replacement back-off scheme. A node can prevent repeatedly decrypting these seats by keeping track of the hashes of each seat previously received and decrypted. The book-keeping of the list of hashes is reduced by regularly deleting the list after a time-out lapses. The performance to search for a hash value can be improved by using a fast access container, such as a hash table.

4.6 Overview of Practical Buses Protocol

The overall operation of the Practical Buses protocol can be described effectively from two different perspectives. One important perspective is the life cycle of a message and the other important perspective is transformation of a bus from a node as it handles the receipt, modification, and forwarding of the bus.

The life cycle of a message involves the creation of a nested encryption of the message, encapsulation in a seat, and processing. The creation of the initial nested encrypted message starts with choosing a random indirection path of random but bounded length. The inner core consists of the actual message prepended with an anonymous public key (required for the receiver to send an acknowledgment) and a message tag field. When adding a layer of encryption, a random AES key in the salt field and a forward flag (true for every layer except the inner core) are prepended to the current message content. The salt is encrypted with RSA-OAEP and the rest of the message is encrypted with AES using the value in the salt field as the key. Encapsulation in a seat consists of appending the nested encrypted message to the seat owner's public key, the seat tag (unique seat identifier), and the seat timestamp, and appending random data to make the seats uniform in size. Thereafter, a digital signature is calculated on the seat with the owner's private key and appended to the seat. A node processing a nested encrypted message contained in a seat first tries to decrypt the salt field with its RSA-OAEP private key and anonymous private key to extract the AES key. This decryption is successful if the result has the prescribed redundancy, in which case the node removes the outermost encryption layer by decrypting the rest of the data with the extracted AES key. After decrypting, the forward field is extracted. If the forward flag is true, then it removes the salt (and the forward fields) and forwards the rest of the decrypted inner layer. If the forward flag is false, the node extracts the message from the inner core. In all cases, the inner core is eventually received by the intended recipient.

The general operation of the protocol consists of a node receiving the bus, modifying it as required to send or receive messages, and then forwarding it to the next node in the network. Receiving and forwarding the bus are straightforward. Receiving the bus consists of caching the entire bus into a receive buffer and checking the bus for any messages intended for that node by decrypting all the salt fields using RSA-OAEP and checking for the prescribed redundancy. For each decrypted salt field with the prescribed redundancy, the AES key is extracted and the rest of the seat is decrypted. If a message's forward flag is false, then the contents of the decrypted seat are passed to the application layer and an anonymous acknowledgment message is created and placed in a queue (the send queue) containing messages to be sent on the bus at the next available opportunity. If the forward flag is true, then the contents of the decrypted message are inserted into the send queue. In both cases, the seat tag is inserted into another queue (the delete queue) containing the tags of messages to be deleted after a random number of bus visits to that node.

Once the received messages have been extracted, the bus is modified to accommodate sending new messages, message forwarding, sending acknowledgments, and seat deletions. The following processing steps are executed:

1. Any seats containing nested encrypted messages that the node created and sent but have not yet been acknowledged are processed. If the number of times the bus has been received with a particular nested encryption has surpassed the threshold r for that message, or if the acknowledgment time-out has expired, a new indirection path is chosen and a new nested encrypted message is created on the reversed indirection path. The seat containing the old nested message is deleted, ensuring that not more than pk seats are deleted. In addition, maximum resend time-outs are checked, and if any are expired the node stops trying to deliver the nested encrypted message and instead delivers an error message to the application. Before inserting a new nested encrypted message to be resent, this step must ensure that it does not use more than $k - pk$ reserved (containing a valid message) seats; otherwise, the node will not be able to delete enough seats in p-threshold replacement back-off scheme. If the node is unable to resend the message, it waits until the next pass of the bus and tries again.

2. Any entries in the delete queue whose random delay has expired are deleted. The total number of seats deleted in this step and the previous step must not

exceed pk for a single pass of the bus — any additional seats ready for deletion are deleted in the next pass of the bus.

3. Any expired seats whose timestamps have expired are deleted. The total number of seats deleted in this step and the previous two steps must not exceed pk for a single pass of the bus.

4. If fewer than pk seats have been deleted, seats that are not reserved should be randomly deleted until a total of pk seats, including those deleted in the previous three steps, are deleted.

5. As many messages in the send queue as possible (these can be resends of unacknowledged messages, forwarded nested encrypted messages, acknowledgments, or new messages) are placed in seats on the bus. At most $k - pk$ seats can be reserved with valid messages in this step; otherwise, the node may not be able to delete sufficient seats in the next iteration. New messages are placed on the bus last, so that existing messages from other nodes are forwarded, and the node does not monopolize the network.

6. If fewer than $k - pk$ messages have been placed on the bus, random data is inserted into seats so that the total number of modified seats for this iteration is $k - pk$.

4.7 Summary

The Practical Buses protocol is the first anonymous communication scheme to preserve mutual anonymity, be secure against all known attacks, provide scalability, and support a dynamic topology. In addition, the Practical Buses protocol is designed to enhance security and anonymity, as well as improve fault tolerance and performance.

The design of the Practical Buses protocol is based upon the optimal buffer complexity Buses protocol in [4]. However, numerous changes have been made to the original Buses protocol to preserve mutual anonymity, be secure against all known attacks, provide scalability, and support a dynamic topology. To provide scalability and support for a dynamic topology while preserving mutual anonymity, three new techniques were designed and incorporated into the Practical Buses protocol: nested encryption with indirection, p-threshold-replacement back-off scheme, and randomly delayed seat deletion. To protect against all known attacks, replay protection is included in the Practical Buses protocol.

To enhance security and anonymity, as well as improve fault tolerance and performance, additional techniques are added to the Practical Buses Protocol. Security and anonymity are enhanced with the addition of anonymous acknowledgments to allow the receiver to acknowledge a received message anonymously and the addition of signed bus seats to prevent seat tampering. Fault tolerance is improved to recover from lost messages by adding resends and to recover from lost buses by adding bus loss tolerance. Performance is improved by speeding up the encryptions/decryptions with hybrid encryption, allowing a user to specify how many seats it owns with dynamic bus seats, the automatic removal of expired bus seats, and reducing the number of seat decryptions required for each bus received by a node.

Chapter 5

Prototype Implementation

This chapter contains the design and description of the prototype implementation of the Practical Buses protocol. A general overview of the prototype is provided in Section 5.1. The main objects in the prototype and their respective header and trailer fields are explained in Section 5.2. The division of the prototype into logical threads and shared objects is explained in Section 5.3. The tunable parameters in the Buses protocol are summarized in Section 5.4. An example of how the protocol works is provided in Section 5.5. Lastly, the prototype is summarized in Section 5.6.

5.1 Prototype Overview

Each node runs a *bus server*, a realization of the Practical Buses protocol, that provides an interface for anonymous communication. The prototype implementation is a layer that sits between the TCP layer and the application layer. This position in the protocol stack is chosen because TCP/IP provides a well-known transport/network layer service, reliably delivering packets in order. Currently, application support is only provided for a single chat program that was implemented for testing. Future versions will provide an interface for additional applications to use the anonymous communication prototype transparently.

The implementation of the prototype grew quickly and the final prototype contained over 18,000 lines of C++ code with comments. To keep the prototype practical, unnecessary features were pruned from the prototype during its development. Specifically, dynamic bus creation in Section 4.4.2 and bus fault tolerance in Section 4.4.3 are not supported in the prototype. Also, the prototype deviates from the design in Section 4.3.2, for simplicity, by having each node own one public/private key pair that is used for nested encrypted messages, as well as for digital signatures and as the anonymous public key. All other design features in Chapter 4 were incorporated into the prototype.

The modular encryption components that the prototype uses are RSA-OAEP [21] with 1024-bit keys, CBC-AES [27] with 128-bit keys, and RSA-SSA (RSA Signature Scheme with Appendix) [22] with 1024-bit keys. These components were chosen because they are fast and have withstood vigorous security analyses. The RSA-OAEP and RSA-SSA components were implemented based on the design documents [21] and [22], respectively. For SHA-1, a subcomponent of RSA-SSA and RSA-OAEP, public-domain software was used from RFC 3174 [36]. The CBC-AES component uses public-domain software written by Stefanek [40]. The GNU Multiple Precision arithmetic library (GMP) version 4.1.1 [16] is used by the encryption components for multi-precision arithmetic. GMP was chosen because of its emphasis on fast implementation, support for multiple CPUs, and one of its main target applications is cryptographic applications and research.

To centralize the testing parameters, all of the alias-public key mappings, public/private key pairs, routing tables, and hostnames were stored in a central file. The bus servers executing on the nodes are distinguished by their aliases and public/private RSA key pairs. An alias is used to identify a receiver anonymously, so that the sender does not have to enter the entire public key of the receiver when constructing a chat message.

Obviously, the storage of public/private key pairs in a central table is not secure. This approach is used only to facilitate the testing of the protocol. In a real deployment, each bus server must securely store its own public/private key pair. Each bus server would then learn all of the public keys when it receives the bus and extracts the unique seat owner fields. This public key pool could then be used when constructing an indirection path. A node wishing to communicate with an anonymous party would have to learn a public key in an anonymous manner (e.g., anonymous public key and associated alias posted on a Web page).

The setup involves updating the alias-public key mappings, updating the routing table with the domain names of all the machines in the test, inserting public/private key pairs into the key table, and adjusting the parameters to reflect the desired configuration. Each bus server is mapped to one of the aliases to determine its public/private key pair, after which it is compiled to reflect the updates. The bus servers are started sequentially on each machine. The bus server corresponding to the last entry in the alias-public key mapping table is started last, so that it can create the bus and launch it on the network. Once the bus server is started, the user can send an anonymous message by entering "receiver message" followed by enter, where receiver is the alias of the anonymous receiver and message is the message to be sent to the receiver. The optional command line arguments for a bus server are:

- -tS sendTraceFileName: This creates a trace of all messages sent by the bus server. The trace is recorded in the file with name sendTraceFileName. Each line in the trace file contains a receipt of a message sent by the bus server:

timestamp, message, message tag, and receiver alias. All fields are separated by spaces.

- -tR recvTraceFileName: This creates a trace of all messages received by the bus server. The trace is recorded in the file with name recvTraceFileName. Each line in the trace file contains a receipt of a message received by the bus server: timestamp, message, and message tag.

- -tA ackTraceFileName: This creates a trace of all messages acknowledged by the bus server, recorded in the file ackTraceFileName. Each line in the file trace contains a receipt of an acknowledgment received by the bus server: timestamp and message tag.

- -w workloadFileName: This changes chat program input to come from the file workloadFileName instead of standard input (stdin). This allows for repeatable tests and the generation of workloads. For example, workloads can be generated *a priori* with a Poisson arrival process.

5.2 Main Objects

The metaphor of a city bus was realized in a hierarchy of three main objects: bus, seat, and nested message object. The bus object contains multiple seats, and each seat object contains exactly one nested message object (may or may not be a valid nested message). The division of the bus object into these three objects simplified the code and followed the modular design in Chapter 4. In addition, the division is logical because each of these objects has a different life cycle and is processed differently.

5.2.1 Nested Message

The nested message class performs two main operations: creation of a nested encrypted message, and decryption of a nested encrypted message to remove one layer of encryption. In order for a sender to create a nested encrypted message, the nested message class constructor requires the public key of the receiver, the message for the receiver, the message size, a message tag (a random 128 bit number), and a set of possible indirection public keys. The constructor then chooses a random indirection path of random bounded length, and iteratively appends a header to each layer and hybrid-encrypts the layer using the public keys of the reversed indirection path and randomly generated 128-bit AES keys. In order for a node receiving a bus to determine if a nested message contained within a seat is destined for the node, the nested message class requires the private key of the node. The hybrid-decryption succeeds when the private key decryption of the encrypted symmetric key has the correct redundancy. The function call returns true and the nested message object is updated with its decryption. Otherwise, the decryption fails because the correct redundancy does not exist and false is returned. Let the notation "nested message" imply a nested encrypted message for the rest of this thesis.

If the nested message is not the innermost core, each layer of nested encryption has a 144-bit nested message header containing four fields: salt, type, forward, and message length. The inner core appends to the header two additional fields: message tag and anonymous PK (see Table 5.1). The inner core nested message header is (272 + size of anonymous public key) bits long.

Table 5.1: Nested Message Header

salt (128 bits)	type (3 bits)	forward (1 bit)
message length (12 bits)	message tag (128 bits)	Anonymous PK (size of public key bits)

The purpose of each nested message header field is as follows:

- The *salt* field contains a random 128-bit symmetric key for the purposes of hybrid encryption. This approach improves performance and ensures that a message resent on a new indirection path produces a different nested message. To further improve performance, the public key encrypts the largest possible plaintext that constitutes a block (e.g., 87 bytes for the RSA-OAEP encryption module with 1024 bit keys) rather then just encrypting only the salt (16 bytes). Thus, the symmetric key encryption continues to encrypt the plaintext starting where the public key encryption left off.

- The *type* field is a 3-bit value used to identify the type of message contained in the nested message structure. Two values are currently permitted:

    1. 000_2 — application nested message: contains an application message.
    2. 001_2 — acknowledgment nested message: contains an acknowledgment where the message is null and the message tag field contains the message tag of the message being acknowledged. This type notifies a sender not to send an acknowledgment after receiving an acknowledgment.
    3. All other bit combinations are reserved for future development. If they are encountered, the nested message is not processed further.

- The *forward* field informs a node if the message contained within the nested message is destined for the node or to be forwarded as a nested message. The former is the case when the forward field is false and the latter is the case when forward is true. In addition, if the forward field is false, the header has two additional fields: message tag and anonymous public key. The forward field is currently interpreted when the nested message is an application or acknowledgment nested message.

- The *message length* field indicates the actual message size in bytes. It is relevant only when the seat type is application. It allows a node to determine where the message ends and where random data begins.

- The *message tag* field of an application message uniquely identifies a message. The message tag of an acknowledgment message uniquely maps the acknowledgment to a message sent. A sender of an application message creates a message tag by generating a random 128 bit number.

- The *anonymous PK* field contains the anonymous public key of the sender. The receiver uses it to create an anonymous acknowledgment to the innermost core of an application message. The sender uses an anonymous public key different from the public key used in the owner field of a seat to prevent the receiver from correlating the public key to the sender.

All fields in each header are encrypted when the layer is encrypted, in order to preserve anonymity. Salt is encrypted because it contains the secret symmetric key used to encrypt the rest of the message. The forward field is encrypted; otherwise an adversary could identify a receiver when a nested message sent to them has the forward field set to false. To prevent size correlation attacks the message length is encrypted. To prevent marker attacks the message tag is encrypted. Lastly, the anonymous public key field is encrypted, otherwise sender anonymity would be defeated when the nested message is first sent by the sender.

5.2.2 Seat

The seat class has one main purpose — to encapsulate a nested message as a seat to place on the bus. The constructor requires the node's public key, the nested message, and the node's private key. The seat class encapsulates a nested message to construct a seat by prepending a seat header, appending random data to make the seat uniform size, and appending a seat trailer.

Each seat header is (160 + size of public key) bits long containing three fields: owner, tag, and timestamp (see Table 5.2). These are sent as plaintext.

Table 5.2: Seat Header

owner (size of public key bits)	tag (127 bits)	timestamp (33 bits)

The purpose of each seat header field is as follows:

- The *owner* field contains the public key of the owner. The field identifies which seats belong to each node.
- The *tag* field is used to identify each seat uniquely for purposes such as seat deletion. A tag is randomly generated upon seat construction.

- The *timestamp* field is the number of seconds since midnight January 1970 (UTC time). It is used to determine if a seat's time-out has expired, resulting in deletion. Also, it is used to provide replay protection.

It is not necessary to encrypt the owner, tag, and timestamp fields because they do not provide an adversary with any information that reduces anonymity. Owner, tag, and timestamp only indicate the predecessor node in the indirection path, not the sender. An adversary eavesdropping on a node can determine the owner and timestamp by correlating which seats are replaced by which nodes and when. The tag is a short identification for a seat. If it were encrypted, the adversary could just record the hash value of each seat as a short identification for each seat.

Seat Trailer

Each seat has a trailer containing a digital signature (see Table 5.3).

Table 5.3: Seat Trailer

signature (size of digital signature bits)

The signature field contains a signature of three items: the seat header, nested message, and random data appended to the nested message to make the seat uniform size. The signature is used to determine if a seat or signature has been altered in transit by someone other than the owner of the seat.

5.2.3 Bus

The bus class has three main purposes:

1. Bus creation and resetting — A node can create a bus consisting of a bus header. When a node receives a bus, it resets its bus object to be the received bus.

2. Seat insertion and deletion — These functions are necessary for the p-threshold replacement back-off scheme. The interface allows the insertion of a nested message, which is encapsulated in a seat by the bus class automatically. The tag of a seat must be provided to the interface to delete a seat, which returns true on successful deletion and false otherwise.

3. Seat extraction — A node that receives a bus extracts all the seats that are destined for it. Each seat is checked for the correct OAEP decryption redundancy, and its signature is verified. An error message is sent to standard error (stderr) if the signature is incorrect, making the user aware that either there are network errors occurring below the TCP layer, or an active adversary has tampered with the signature or seat contents. Thereafter, a tampered seat is deleted and ignored.

The bus header is 24 bits long and consists of two fields: *bus type* and *number of seats* (see Table 5.4).

Table 5.4: Bus Header

bus type (8 bits)	number of seats (16 bits)

The purpose of each bus header field is as follows:

- *bus type* — informs a node of what type of message to expect, making the node aware of the message format. The current valid bus types are:
    1. 00000000_2 — control message. A control message is used to denote a leader election message. This feature is not supported in the prototype implementation.
    2. 00000001_2 — bus message. A bus message denotes a bus being sent.
    3. All other bit combinations are reserved for future developments.
- *number of seats* — This field informs a node receiving a bus of the bus size.

5.3 Multi-threading

The bus server is multi-threaded using Pthreads [23], a multi-threaded library that is simultaneously an IEEE Standard, an ISO/IEC Standard, and an Open Group Technical Standard [32]. Pthreads was chosen because it is supported by most hardware vendors in their proprietary application programming interface. The bus server consists of a *Bus thread*, a *Read thread*, a *Write thread*, and an *Acknowledgment thread*. These threads provide a logical separation of the work performed by the bus server, and will facilitate parallel execution of the prototype on shared-memory multiprocessors in future work. Section 5.3.1 contains an overview of all the threads. Then, in Section 5.3.2 the main thread objects are identified. For the interested reader, the pseudo-code for the Bus thread, Read Thread, Write thread, and Acknowledgment thread is provided in Appendices B.1, B.2, B.3, and B.4, respectively.

5.3.1 Thread Overview

The four threads in the bus server operate as follows. The Bus thread receives the bus from the network, modifies it as needed, and forwards it to the next hop on the route. As part of its operation, the Bus thread must store (cache) the bus temporarily for the Read thread to process it. The Read thread checks the cached bus for valid seats, and either delivers messages locally for display to the user, or forwards an appropriately decrypted inner layer to the next hop, as required. In addition, the Read thread schedules randomly-delayed seat deletions for the Bus thread, and passes acknowledgment information to the Acknowledgment thread. The Acknowledgment thread creates acknowledgments for the Bus thread to send. The Write thread handles user input. It parses chat program messages entered by the user, and creates a nested message that is queued for the Bus thread to process.

5.3.2 Main Thread Objects

The main thread objects are used to communicate among the threads and keep track of state. For each object, its purpose, dependencies, and shared state are explained:

1. *busQueue* (shared) — The Bus thread stores a received bus for later retrieval by the Read thread.

2. $forwardAckQueue$ (shared) — This queue contains application nested messages that the Read thread inserts, and acknowledgment nested messages that the Acknowledgment thread inserts. The Bus thread extracts these nested messages, deleting them from the $forwardAckQueue$, and sends them in seats via the p-threshold replacement back-off scheme, recording their corresponding

tags in the *reservedTagList* (described below).

3. *sendQueue* (shared) — This queue contains nested messages inserted by the Write thread. The Bus thread extracts these nested messages, deletes them from the *sendQueue*, and sends them in seats via the p-threshold replacement back-off scheme. The sent seat tags are inserted into the *reservedTagList*. The *sendQueue* entries differ from the *forwardAckQueue* entries because the necessary information to disguise or resend the seat (e.g., unacknowledged message) also appears in the *sentList*.

4. *sentList* (shared) — The purpose of this list is to keep track of sender-generated nested messages and acknowledgments for the purposes of disguising and resending a message that was not acknowledged. The Bus thread inserts information into the *sentList* when it removes a seat from the *sendQueue*. The following items are recorded in each element of the *sentList*:

   (a) The seat tag of the seat that the nested message was sent in.

   (b) The number of times the nested message has been resent since the last reset of r (initialized to 1). r is the individual replacement threshold in the p-threshold replacement back-off scheme.

   (c) The value of r for that nested message.

   (d) The timestamp when the message was first sent.

   (e) The timestamp when the message was last sent/resent by the Bus thread.

   (f) The value of the message tag, message, message length, and the receiver's public key.

In addition, the Bus thread updates the *sentList* entries every time it receives the bus by checking each tag in the *sentList* with the bus. The Read thread is responsible for removing each entry via the message tag upon receipt of an acknowledgment. The only other way a *sentList* entry is removed is by the Bus thread when the maximum send time has elapsed.

5. *reservedTagList* (not shared) — This list is used by the Bus thread to store tags of reserved seats, inserted by the Bus thread. Each time the Bus thread receives the bus, it deletes any of the tags in the *reservedTagList* that are no longer on the bus. The Bus thread then uses the list to determine which seats are eligible to be replaced.

6. *delTagList* (shared) — The list contains 2-tuple entries {tag, delay} of seats to be deleted. The Read thread places the 2-tuple in the list. The delay is a random but bounded delay for seat deletion. The Bus thread processes the list every time it receives the bus. The delay is decremented by 1, and when $delay \leq 0$ the seat with the corresponding tag is deleted, provided that the p-threshold-replacement back-off scheme allows it.

7. *receivedMessagesList* (not shared) — This list is used by the Read thread to record the 3-tuple {salt, timestamp, embedded nested message} of any nested messages received with the forward field true. The 3-tuples are recorded in the list within a time window to prevent replays. To prevent duplicate messages arriving at the receiver, if the forward field in the nested message is false then the message tag is recorded instead of the salt as the first entry in the 3-tuple.

8. *ackDataQueue* (shared) — The Read thread writes the 2-tuple {anonymous PK, message tag} to this queue. The Acknowledgment thread extracts the 2-tuple to create an acknowledgment nested message.

9. *seatHashTable* (not shared) — The Read thread prevents decrypting repeated seats, by checking if the hash of a seat is already in the *seatHashTable*. If it is not in the *seatHashTable*, it inserts the hash of the seat and proceeds to decrypt it. Otherwise, it ignores the repeated seat.

5.4 Parameters for the Bus Server

Each bus server has a set of parameters:

1. *busServerPort* — The port number used for a bus server forwarding a bus to connect to another bus server to receive the bus. All bus servers must use the same *busServerPort*.

2. N — number of bus servers in static routing table.

3. k — number of seats that the node owns.

4. K — upper-bound on k to bound worst case buffer complexity.

5. p — the percentage of seats being randomly replaced in the p-threshold replacement scheme. Note that $\lfloor pk \rfloor$ is used when the product is not an integer.

6. *uniformSeatSize* — the uniform size in bytes of all seats inserted by the bus server.

7. $maxBusSize$ — the maximum input buffer size in bytes that a bus server will accept without memory error. For a static network with N nodes, this value should always be $((N * K * uniformSeatSize)/8)$ bytes.

8. $maxOldSeat$ — specifies the time-out before a seat is deleted.

9. $maxDuplicateTime$ — specifies the time window for which nested messages are recorded to prevent replay messages.

10. $ackTimeOut$ — specifies how long until a message is resent due to absence of receiver acknowledgment.

11. $maxResendTime$ — specifies how large a time window to spend retransmitting before the bus server gives up and reports an error message to the application.

12. D — range of $[0, D-1]$ for randomly delayed seat deletion.

13. L — range of $[1, L]$ for random indirection path length.

14. $staticL$ — Set to 1 to have exactly L layers around every inner core. Set to 0 to have the number of random layers around the inner core in $[1, L]$. This parameter is set to 1 for scalability analysis of indirection layers; however, it should be set to 0 for optimal anonymity.

15. R — the initial number of times each sender message can be resent without being disguised. This initial value for each sender message r may be increased by the p-threshold replacement back-off scheme.

16. $busThreadWait$ — set to 1 to prevent the Bus thread from forwarding the bus until the Read thread has removed the last entry from the $busQueue$.

Otherwise, the Bus thread does not wait at all to forward a bus. This parameter must be set to 1 when decryption of all the seats takes longer on average than it takes for the bus to perform one bus tour. Otherwise, the *busQueue* will grow indefinitely and message latency will increase indefinitely.

17. $verifyAllSeats$ — set to 1 to check signatures on all seats. Otherwise, set to 0 to verify only seats that are valid (correct redundancy in decryption). To allow a node to determine when a message has been tampered with, set to 1. Otherwise, for better performance, set to 0. Note that setting to 0 does not reduce anonymity but allows for a denial of service (DoS) attack which corrupts all seats so that their decryption does not have the correct redundancy.

18. $randomWalk$ — set to 1 to have the bus server randomly choose the neighbour to which the bus is forwarded. Otherwise, set to 0 to have bus server use the next entry in the routing table.

19. *reduceDecrypt* — set to 1 to have the bus server keep track of previously received seats to reduce seat decryptions. Before decrypting a seat, the bus server checks if it was previously received. To disable, set to 0.

20. $reduceDecryptInterval$ — specifies the time interval in seconds for the chained hash table $seatHashTable$ to be cleared.

Note that the parameters $k, p, uniformSeatSize, maxOldSeat, maxDuplicateTime,$ $ackTimeOut, maxResendTime, D, L, staticL, R, verifyAllSeats, reduceDecrypt,$ and $reduceDecryptInterval$ can be tuned to improve performance. However, the parameters $maxDuplicateTime, ackTimeOut, D, L, staticL, R$ can all affect anonymity if set incorrectly, as discussed in Section 6.2.

5.5 Practical Buses Example

To illustrate the operation of the Practical Buses protocol, consider the following scenario. A sender S wants to send the message ‘Hello’ to a receiver R.

5.5.1 Creation of nested message by S’s bus server’s Write thread

1. The Write thread receives ‘Hello’ from the application.
2. The Write thread chooses a dynamic path length l where $l \leq L$. Suppose $l = 2$ and the random indirection path is K_1^+, K_2^+, K_R^+. Let the nodes be denoted $P_1, P_2,$ and R, respectively, and S’s anonymous public key by $K_{S_{anon}}^+$.
3. The Write thread creates a nested message L_1 using the reverse of the indirection path by creating L_R, L_2, L_1 as follows:

   $$MessageHeader_R = (salt_R, 000_2, 0, 6, msgTag, K_{S_{anon}}^+)$$
   $$L_R = E_{K_R^+}(MessageHeader_R, \text{‘Hello’}).$$

   Note that since forward is false in $MessageHeader_R$, there are the additional fields message tag and an anonymous PK, where message tag contains a random 128-bit number.
   L_2 and L_1 are created as follows:

   $$MessageHeader_2 = (salt_1, 000_2, 1, sizeOf(L_R))$$
   $$L_2 = E_{K_2^+}(MessageHeader_2, L_R)$$

   $$MessageHeader_1 = (salt_2, 000_2, 1, sizeOf(L_2))$$
   $$L_1 = E_{K_1^+}(MessageHeader_1, L_2).$$

Each $MessageHeader$ has different random salt, and its message size is adjusted according to the message being sent. Also note that the seat type is an application seat (000_2) for all of the messages and the forward bit is set to true (1) for every header except for $MessageHeader_R$ intended for the receiver.

4. Bus thread places L_1 into the $sendQueue$.

5.5.2 Placement of L_1 onto the Bus by S's Bus thread

1. Eventually the Bus thread extracts L_1 from the $sendQueue$ and records the appropriate information to the $sentList$ (used later for disguising a message and processing receiver-end acknowledgments).
2. The Bus thread saves the incoming bus to $busQueue$.
3. The Bus thread prepends the seat header, adds random data x_1 to L_1 so that the seat is of uniform size, and appends a trailer to L_1 to create $Seat_S$ as follows:

$$SeatBody_S = (L_1, x_1)$$
$$SeatHeader_S = (K_S^+, tag_S, timestamp_S)$$
$$SeatTrailer_S = (signature(SeatHeader_S, SeatBody_S))$$
$$Seat_S = (SeatHeader_S, SeatBody_S, SeatTrailer_S).$$

4. The Bus thread places $Seat_S$ on the Bus and forwards the bus.
5. If the bus is received r times before the next hop in the indirection path acknowledges it (by deleting the seat), then a new nested message is created with a new indirection path but the same message tag, anonymous PK, and message and encapsulated in a seat. In addition, its entry in $sentList$ is updated.

6. If the receiver-end acknowledgment times out ($ackTimeOut$), then it is disguised and resent. A new nested message is created with a new indirection path but the same message tag, mutual PK, and message. If $maxResendTime$ elapses, then an error is reported to the application that the message could not be delivered. In addition, the corresponding entry in $sentList$ is updated.

5.5.3 Receipt of the seat by P_1 (owner of K_1^+)

1. The Bus thread saves the incoming bus to $busQueue$.
2. Eventually the Read thread decrypts $SeatBody_S$. That is, it decrypts the nested message salt field with its private key, checks for the correct OAEP redundancy, and verifies the signature. If the correct OAEP redundancy is not present, then the nested message is disguised and eventually resent by the sender due to lack of acknowledgment. If the signature verification does not pass, then an error message is passed to the application making the user aware that either there was a network error or a malicious adversary is tampering with seats. In addition, the nested message is not further processed. The decryption of $SeatBody_S$ yields
$$D_{K_1^-}(SeatBody_S) = (MessageHeader_1, L_2).$$
3. The Read Thread places the 2-tuple $\{tag, rand(1, L)\}$ into $delTagList$ for the Bus thread to process and eventually delete the seat.
4. The Read Thread places L_2 into the $forwardAckQueue$ because the forward bit is set to 1 in the $MessageHeader_1$.
5. Eventually, after a few rounds of the bus, the Bus thread extracts L_2 from the $forwardAckQueue$.

6. The Bus thread prepends the seat header, adds random data x_2 to L_2 so that the seat is of uniform size, and appends a trailer to the seat and creates $Seat_1$ as follows:

$$SeatBody_1 = (L_2, x_2)$$
$$SeatHeader_1 = (K_1^+, tag_1, timestamp_1)$$
$$SeatTrailer_1 = (signature(SeatHeader_1, SeatBody_1))$$
$$Seat_1 = (SeatHeader_1, SeatBody_1, SeatTrailer_1).$$

7. The Bus thread places $Seat_1$ on the Bus and forwards the bus.
8. Until the next hop in the indirection path acknowledges the seat (by deleting it), P_1 keeps resending the seat and never disguises it via the p-threshold replacement back-off scheme.

5.5.4 Receipt of the seat by P_2 (owner of K_2^+)

1. The Bus thread saves the incoming bus to *busQueue*.
2. Eventually the Read thread decrypts $SeatBody_1$. That is, it decrypts the nested message salt field with its private key, checks for the correct OAEP redundancy, and verifies the signature. If the correct OAEP redundancy is not present, then the nested message is disguised and eventually resent by the sender due to lack of acknowledgment. If the signature verification does not pass, then an error message is passed to the application making the user aware that either there was a network error or a malicious adversary is tampering with seats. In addition, the nested message is not further processed. The decryption of $SeatBody_1$ yields

$$D_{K_2^-}(SeatBody_1) = (MessageHeader_2, L_R).$$

3. The Read thread places the 2-tuple $\{tag, rand(1, L)\}$ into $delTagList$ for the Bus thread to process and eventually delete the seat.
4. The Read thread places L_R into the $forwardAckQueue$ because the forward bit is set to 1 in the $MessageHeader_2$.
5. Eventually, after a few rounds of the bus, the Bus thread extracts L_R from the $forwardAckQueue$.
6. The Bus thread appends the seat header, adds random data x_R to L_R so that the seat is of uniform size, and appends a trailer to the seat and creates $Seat_2$ as follows:

$$SeatBody_2 = (L_R, x_R)$$
$$SeatHeader_2 = (K_2^+, tag_2, timestamp_2)$$
$$SeatTrailer_2 = (signature(SeatHeader_2, SeatBody_2)$$
$$Seat_2 = (SeatHeader_2, SeatBody_2, SeatTrailer_2).$$

7. The Bus thread places $Seat_2$ on the Bus and forwards the bus.
8. Until the next hop in the indirection path acknowledges the seat (by deleting it), P_2 keeps resending the seat and never disguises it via the p-threshold replacement back-off scheme.

5.5.5 Receipt of the seat by receiver (owner of K_R^+)

1. The Bus thread saves the incoming bus to $busQueue$.
2. Eventually the Read thread decrypts $SeatBody_2$. That is, it decrypts the nested message salt field with its private key, checks for the correct OAEP redundancy, and verifies the signature. If the correct OAEP redundancy is not present, then the nested message is disguised and eventually resent by the

sender due to lack of acknowledgment. If the signature verification does not pass, then an error message is passed to the application making the user aware that either there was a network error or a malicious adversary is tampering with seats. In addition, the nested message is not further processed. The decryption of $SeatBody_2$ yields

$$D_{K_R^-}(SeatBody_2) = (MessageHeader_R, \text{`Hello'}).$$

3. The Read thread places the 2-tuple $\{tag, rand(1, L)\}$ into $delTagList$ for the Bus thread to process and eventually delete the seat.
4. The Read thread places the 2-tuple $\{\text{anonymous PK}, \text{message tag}\}$ into the $ackDataQueue$, where $\{\text{anonymous PK}, \text{message tag}\}$ are extracted from the $MessageHeader_R$.
5. The Acknowledgment thread creates an acknowledgment nested message. Its creation is similar to the generation of the sender's nested message, except the inner core has the same message tag as the message being acknowledged, the inner core is encrypted with the anonymous public key contained in the anonymous PK field of the sender's message, the nested message type is acknowledgment (001_2), and the message is NULL. The acknowledgment nested message is inserted into the $forwardAckQueue$.
6. The Read thread passes message 'Hello' to application since the forward bit is 0 in $MessageHeader_R$.
7. Eventually, the Bus thread sends the acknowledgment nested message. Note that the forwarding of the acknowledgment nested message is identical to the process described above to forward the application nested message.

5.5.6 Receipt of the Acknowledgment by S

1. Eventually, the encrypted inner core of the acknowledgment nested message is received by S.
2. Eventually the Read thread decrypts the acknowledgment nested message. That is, it decrypts the nested message salt field with its anonymous private key, checks for the correct OAEP redundancy, and verifies the signature. If the correct OAEP redundancy is not present, then eventually a new acknowledgment will be sent in response to the sender's disguised message being re-sent due to lack of acknowledgment. If the signature verification does not pass, then an error message is passed to the application making the user aware that either there was a network error or a malicious adversary is tampering with seats. In addition, the nested message is not further processed. The decryption of the acknowledgment nested message yields

$$D_{K_R^-}(acknowledgmentSeat) = (MessageHeader_{ack}, NULL).$$

3. The Read thread places the 2-tuple $\{tag, rand(1, L)\}$ into $delTagList$ for the Bus thread to process and eventually delete the seat.
4. The Read thread removes the message with the corresponding message tag (extracted from $MessageHeader_{ack}$) from $sentList$ because the nested message type was acknowledgment.

5.6 Summary

The Practical Buses prototype is an implementation based upon the design in Chapter 4, except dynamic bus creation and bus fault tolerance are not supported. The bus server sits between the TCP/IP layer and the application layer in the protocol stack and currently only supports a chat program written for testing. The modular encryption components used are RSA-OAEP (1024-bit keys), CBC-AES (128-bit keys), and RSA-SSA(1024-bit keys).

The prototype is modularized by containing a hierarchy of three main objects and multi-threading the program into four main threads. The hierarchy of objects is a bus that contains multiple seats, and each seat contains exactly one nested message. All of these objects include bit-wise headers and the seat object also includes a bit-wise trailer. The headers and trailers are used for routing a bus containing seats and nested messages through the anonymous practical Buses network. The four main threads are the Bus thread, Read thread, Write thread, and Acknowledgment thread. Each thread is intertwined with the other threads, and a variety of shared objects and mutual exclusion variables are used to communicate among the threads.

To facilitate testing, the administration of the anonymous practical Buses network is centralized and command line arguments are accepted by the bus server. The central files contain numerous parameters, routing information, public/private key pairs, and alias-public key mappings. The command line arguments allow the bus server to produce traces of messages sent, received, and acknowledged as well as read workloads from a file.

Chapter 6

Evaluation

This chapter contains an evaluation of the Practical Buses protocol from two perspectives. First, the Practical Buses prototype's functionality, latency, and scalability are evaluated experimentally in Section 6.1. Second, the Practical Buses protocol's anonymity is evaluated theoretically in Section 6.2.

6.1 Experimental Results

The testing of our Practical Buses protocol uses a simple text-messaging application, similar to chat and electronic mail applications on the Internet. Users can enter messages using their computers, and view messages sent to them by other users. Messages are transmitted securely and anonymously between computers using the Practical Buses protocol. The experimental tests focus on the performance and show that the Practical Buses protocol works well.

6.1.1 Experimental Setup

We used a Beowulf Cluster consisting of 14 identically-configured dual-processor 2.4 GHz Pentium machines. Each machine has 2 GB RAM and a 512 KB cache. The operating system is Red Hat Enterprise Linux Workstation release 3 (Taroon Update 2) 2.4.20-19.7. Each machine in the cluster has a 1 Gbps Ethernet connection to a dedicated Ethernet LAN switch. The prototype uses POSIX threads with libc-2-2.5

and GMP Version 4.1.1, compiled using g++ version 3.2 with flags -g -lgmp -lpthread. To simplify the analysis of the results, the tests were restricted to use only a single CPU on each node and the cluster was isolated so that there was no additional load.

The software setup involves updating the central parameter and routing files. Next, each bus server is recompiled to reflect the updates. Thereafter, the bus servers are sequentially started on each machine. The bus server corresponding to the last entry in the alias-public key mapping table is started last, so that it can create the bus and launch it on the network.

To simplify the presentation of the experimental set-up for each test, assume that the routing table information is only updated once and that the parameter settings default to the values in Table 6.1, unless otherwise specified. The justification for these parameter settings follow. The $busServerPort$ has to be a non-reserved port, that is $busServerPort \geq 1024$. N is the number of nodes in the Buses network. k is the number of seats owned an individual node node. It is restricted so that $\lfloor pk \rfloor > 1$ and $(k - \lfloor pk \rfloor) > 1$. A happy median between bus size and bus tour latency is $k = 4$ for our chat program. K should be the maximal k value used by all the nodes. The parameter p should be no larger than $p = 0.5$. If $p > 0.5$ then a node can insert more seats, but it reduces the number of reserved seats so a node can only insert more garbage. The optimal value of p to handle bursty traffic (e.g., our chat program traffic) is $p = 0.5$ since this allows the node to insert the maximal amount of valid seats than can be reserved. If $p < 0.5$, then the node does not handle bursts as well because it cannot insert as many new seats, but it can sustain a higher throughput because it can have more reserved seats. The $uniformSeatSize$ should be large enough to handle the largest expected message. The maximum message size is $((uniformSeatSize - (3240) - (L * (252)) - 2336)/8 - L * 57)$ bytes which accounts for the seat and nested message headers and trailer, the number of layers L, and the ciphertext inflation due to encryption. The $maxBusSize = ((N * K * uniformSeatSize) + 3)$ bytes, that is the maximum number of seat bytes in a bus plus the size of the bus header. The parameter $maxOldSeat$ should be at least $maxResendTime + RTT + 4RTD$ where RTT is the average measured round trip time for the bus and RTD is the measured standard deviation of the round trip time. RTT and RTD both depend on the number of nodes in the network, the network hardware, and computer hardware at the nodes. RTT is added to prevent a premature seat expiration, and $4RTD$ accounts for the variability. Assuming a normal distribution, this accounts for 95% of the possible events. The parameter $maxDuplicateTime$ should be at least one hour to reduce the threat of replay attacks. The parameter $ackTimeOut$ should be at least $RTT + 4RTD$. $maxResendTime$ should be $x{*}ackTimeOut{+}RTT{+}4RTD$ to guarantee a maximum of x re-sends. L and D should be at least 2 to provide acceptable levels of anonymity. The parameter $staticL$ should be 0 for optimal anonymity. R should be at least 0, and scaled up for improved performance at decreased sender anonymity. $busThreadWait$ should be 1 unless bus processing by the read thread is finished before another bus is received. $verifyAllSeats$ should be 1 to better identify DoS attacks, and 0 for better performance. The parameter $randomWalk$ should be 0 unless the network topology is unknown. $reduceDecrypt$ should be 1 to improve performance, and $reduceDecryptInterval$ should be larger then the time it takes the bus to tour the network but small enough to allow efficient look-up in the $seatHashTable$. For our testing $maxOldSeat$, $maxResendTime$, $ackTimeOut$, and $maxResendTime$ were over inflated to simlify analysis of the experimental results.

To obtain consistent results, workloads are generated *a priori* for each test. The command line argument *-w workloadFileName* is used to read the workloads. To collect experimental timing results, the command line arguments *-tS sendTraceFile-Name*, *-tR recvTraceFileName*, and *-tA ackTraceFileName* are used for each bus server.

For the experiments, the random walk bus forwarding feature was disabled and a statically-configured round-robin tour of the network nodes was used instead. The protocol would not scale well for a dynamic network using a random walk, because it would take an expected time of $O(N^2)$ for the bus to circulate the network once in a completely connected Ethernet network, where N is the number of bus server nodes. However, the results are still relevant because applications for which the network topology is known should clearly make use of that knowledge. Testing and utilizing the dynamic topology functionality to protect against active adversaries and provide additional fault-tolerance will be the subjects of future research.

Table 6.1: Default Parameter Settings

$busServerPort$	1555
N	2
k	4
K	4
p	0.5
$uniformSeatSize$	1024 bytes
$maxBusSize$	((N*K*uniformSeatSize) + 3)
$maxOldSeat$	600 seconds
$maxDuplicateTime$	1000 seconds
$ackTimeOut$	100 seconds
$maxResendTime$	400 seconds
D	2
L	2
$staticL$	0
R	4
$busThreadWait$	1
$verifyAllSeats$	0
$randomWalk$	0
$reduceDecrypt$	1
$reduceDecryptInterval$	60

6.1.2 Latency Tests

The latency tests study the basic operation of the Practical Buses protocol and the impacts of several configuration parameters on its performance. The primary performance metric is *message latency*, defined as the elapsed time between the sender entering a message for the Write thread and the sender receiving via the Bus an acknowledgment of that message's successful delivery. The message delivery time to the receiver is typically half of the message send-ack latency. These experiments use a simple two-node configuration in the cluster, with one node as the sender of messages, and the other node as the receiver. Unless stated otherwise, the message arrival process is deterministic (constantly spaced in time), with fixed size messages.

Figures 6.1, 6.2, and 6.3, illustrate the mean message latency results for each parameter value, with the vertical bars showing one standard deviation above and below the mean. Ten message exchanges were sufficient for each parameter value since the results were consistent, agreed with the theoretical expectations and were reproducible. Figure 6.4 contains a time series plot of the message latency for each message, with message arrival rates of 60, 120, 240, and 360 messages per minute. The number of message exchanges was increased to fifty for this latter test in order to demonstrate more clearly the resulting behavior of an overloaded system.

Figure 6.1 illustrates the impact of message size on message latency. Ten messages, 20 seconds apart, are sent for each message size: 64, 128, 256, 1024, and 4096 bytes. Table 6.2 lists the parameters that are changed from the default parameter settings in Table 6.1.

Figure 6.1 illustrates that for the set of values tested, message latency is indepen-

Table 6.2: Relevant Parameter Settings For Figure 6.1

$uniformSeatSize$	5120 bytes
L	0
$staticL$	1

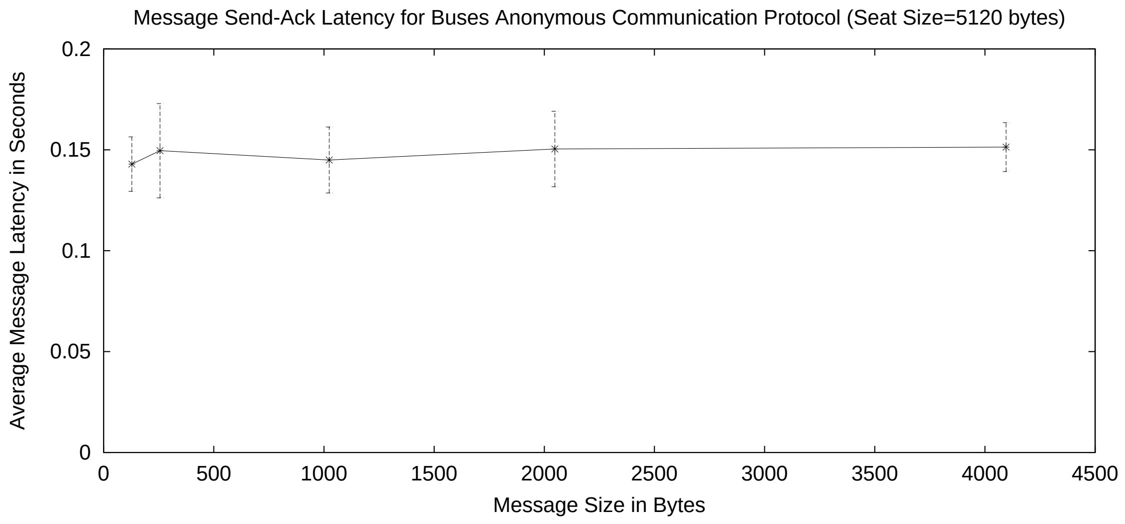

Figure 6.1: Effect Of Message Size In A Simple Two-Node LAN Configuration

dent of message size. The mean message latency is approximately 0.15 seconds. The software RSA decryption and signature verification is the bottleneck of the protocol (this is further investigated later on, see for example Figure 6.7). Each node that receives a bus must hybrid-decrypt all of the seats. For each seat, the node decrypts the AES key with the RSA-OAEP private key. If the correct redundancy is present, the node decrypts the rest of the message with its AES key. Public key decryptions take significantly longer than symmetric key decryptions. In addition, once a seat is found to be valid, its signature is verified, requiring a costly RSA encryption. As a result, the time it takes for the AES decryptions of valid messages do not contribute significantly to the message latency.

As a comparison to the message latency in Figure 6.1, let us consider what the theoretical message latency would be for Figure 6.1 if just the message was sent, and a one byte acknowledgment was received. The sources of network latency are transmission delay, propagation delay, queuing delay, and nodal processing delay. Let us assume that there is no queuing delay (since we would only send one message every 20 seconds over a switched gigabit Ethernet network) and that the nodal processing delay is 0.001 s for each node (the sender, receiver, and switch). We are left with the transmission, propagation, and nodal processing delay of the message being sent and the transmission, propagation, and nodal processing delay of the ensuing acknowledgment. Any data sent must traverse two hops, one from the sender to the switch, and one from the switch to the receiver. Per hop transmission delay is equal to the amount of data sent divided by the rate (10^9 bps). Per-hop propagation delay is equal to the distance of the physical medium divided by speed of propagation. A worst case estimate is a distance of 5 meters between each node and the switch in the cluster, and that the transmission rate is $2.0 * 10^8$ m/s. Thus the propagation delay is $5.0 * 10^{-8}$ s for the message, and $5.0 * 10^{-8}$ s for the acknowledgment. Transmission delay for the acknowledgment is $1.6 * 10^{-8}$ s and $1.024 * 10^{-6}$ s for a 64 byte message, $2.048 * 10^{-6}$ s for a 128 byte message, $4.096 * 10^{-6}$ s for a 256 byte message, $1.6384 * 10^{-5}$ s for a 1024 byte message, and $6.5536 * 10^{-5}$ s for a 4096 byte message. Nodal processing delay is 0.002 s for the message and 0.002 s for the acknowledgment. Thus, the total theoretical message latency for just the message being sent and the ensuing one byte acknowledgment under the assumptions above is dominated by the nodal processing delay. Therefore the theoretical message latency is 0.04 s.

Figure 6.2 illustrates the effect of seat size on message latency. Ten 128 byte messages 20 seconds apart are sent for each uniform seat size: 1024, 2048, 4096, and 8192 bytes. Table 6.3 lists the parameters that were changed from the default parameter settings in Table 6.1.

Table 6.3: Relevant Parameter Settings For Figure 6.2

$uniformSeatSize$	1024, 2048, 4096, and 8192 bytes
L	0
$staticL$	1

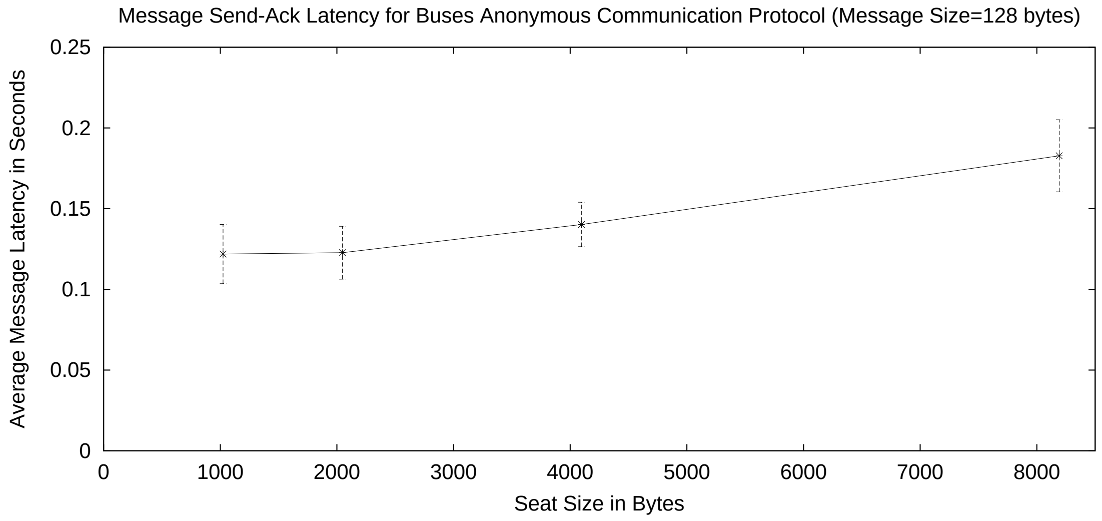

Figure 6.2: Effect Of Seat Size In A Simple Two-Node LAN Configuration

Figure 6.2 shows that for the set of values tested, message latency is linearly dependent on seat size. However, this linear dependence is quite small and the average message latency only increases by 0.06 when the seat size is increased from 1024 bytes to 8192 bytes. Increasing the seat size results in a larger seat, more random data being created to make the seat uniform in size, and a larger bus being sent/received over the network. Sending/receiving a larger seat increase the time it takes for the bus to travel from one node to the next, increasing message latency. However, changing the seat size does not affect the time it takes to decrypt a seat containing a message of the same size nor does the generation of more random data take a significant amount of time.

Figure 6.3 illustrates the effect that the number of indirection layers has on message latency. Ten 128 byte messages 40 seconds apart are sent for each number of indirection layers: 1, 2, 3, 4, and 5. Table 6.4 lists the parameters that were changed from the default parameter settings in Table 6.1.

Table 6.4: Relevant Parameter Settings For Figure 6.3

$uniformSeatSize$	2048 bytes
L	1,2,3,4,5
$staticL$	1

Figure 6.3 shows that message latency increases linearly with the length of the indirection path, as expected. This corresponds with the theoretical analysis because the expected number of bus tours that a bus has to complete before a message is delivered is linearly related to the number of indirection layers. For example, consider the scenario where the sender $node_1$ sends the message m to the receiver $node_2$ and there are $N = 2$ nodes in the network. Let the bus traverse the network as a ring, that is $node_1, node_2, node_1, \ldots$. When the sender chooses a random indirection path of length 5, the indirection path with the final hop to the receiver $node_2$ could consist of $node_2, node_1, node_2, node_1, node_2, node_2$. This results in the bus traversing on average 1.5 nodes to remove each layer of encryption, during three complete bus

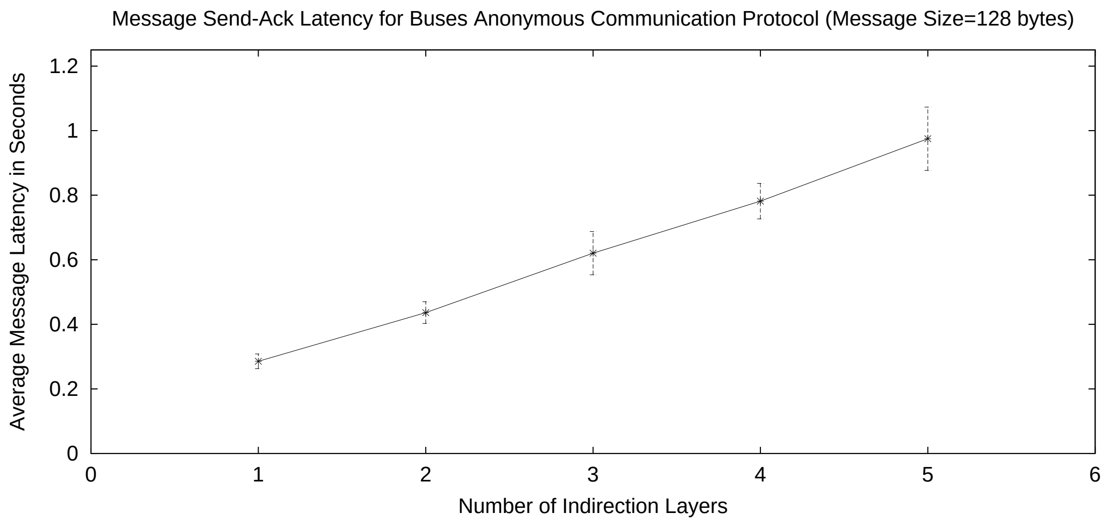

Figure 6.3: Effect Of Number Of Indirection Layers In A Simple Two-Node LAN Configuration

tours and one partial bus tour.

Figure 6.4 illustrates the effect that the message arrival rate has on message latency. Fifty 128 byte messages are sent with message arrival rates of 60, 120, 240, and 360 messages per minute. Table 6.5 lists the parameters that were changed from the default parameter settings in Table 6.1.

Table 6.5: Relevant Parameter Settings For Figure 6.4

$uniformSeatSize$	2048 bytes
$staticL$	1

Figure 6.4 shows the sensitivity to message load. The system performance is steady for message arrival rates of 60 or 120 messages per minute. With a message arrival rate of 240 or 360 messages per minute, a backlog of messages builds up in the queues, increasing per-message latency. This graph shows that the current

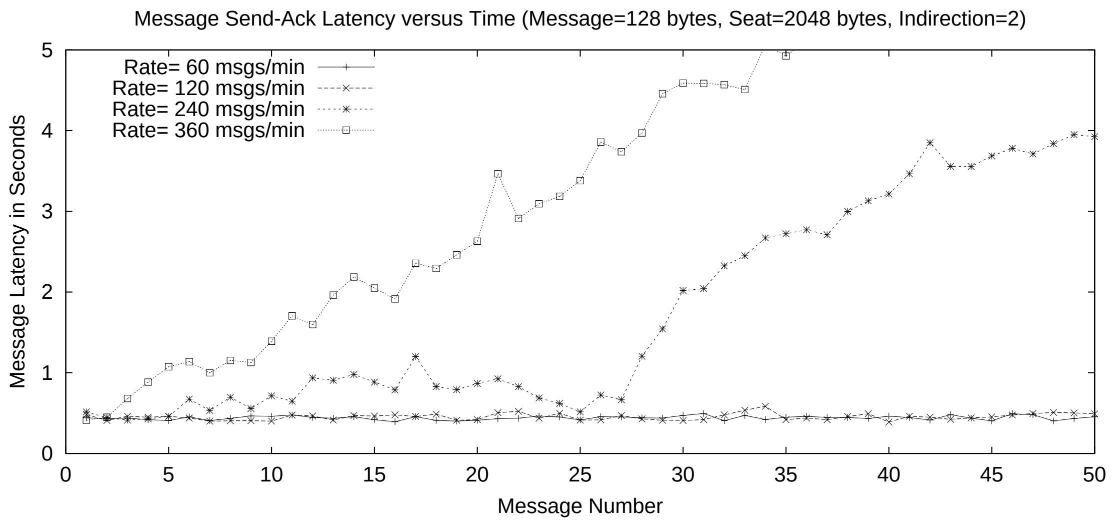

Figure 6.4: Effect Of Message Arrival Rate In A Simple Two-Node LAN Configuration

implementation can sustain an aggregate message arrival rate of at least 120 messages per minute.

6.1.3 Scalability Tests

The second set of experiments considers slightly larger network scenarios on the Beowulf Cluster, to assess the scaling properties of the Practical Buses protocol. For all these tests, the bus circulates in a statically-configured round-robin tour of the network nodes. Empirical analysis of scalability was chosen over theoretical analysis. Theoretical analysis would have been quite complex and would require simplifying assumptions of the model, but these assumptions would make the model too unrealistic.

Figure 6.5 shows the average message latency as the number of nodes in the network is increased from 2 to 8. In this experiment, $node_1$ is always the sender and $node_2$ is always the receiver. The additional nodes in the network form part of the Bus tour, and all nodes are candidate intermediate nodes for indirection. The message arrival rate is ten 128 byte messages per minute. A total of ten messages were exchanged since the results were consistent and agreed with the theoretical expectations and were reproducible. The results plotted in Figure 6.5 show the average message latency for all messages, and the bar above and beneath depicts the standard deviation. Table 6.6 lists the parameters that were changed from the default parameter settings in Table 6.1.

Table 6.6: Relevant Parameter Settings For Figure 6.5

N	3, 4, 5, 6, and 8
$uniformSeatSize$	2048 bytes
$staticL$	1

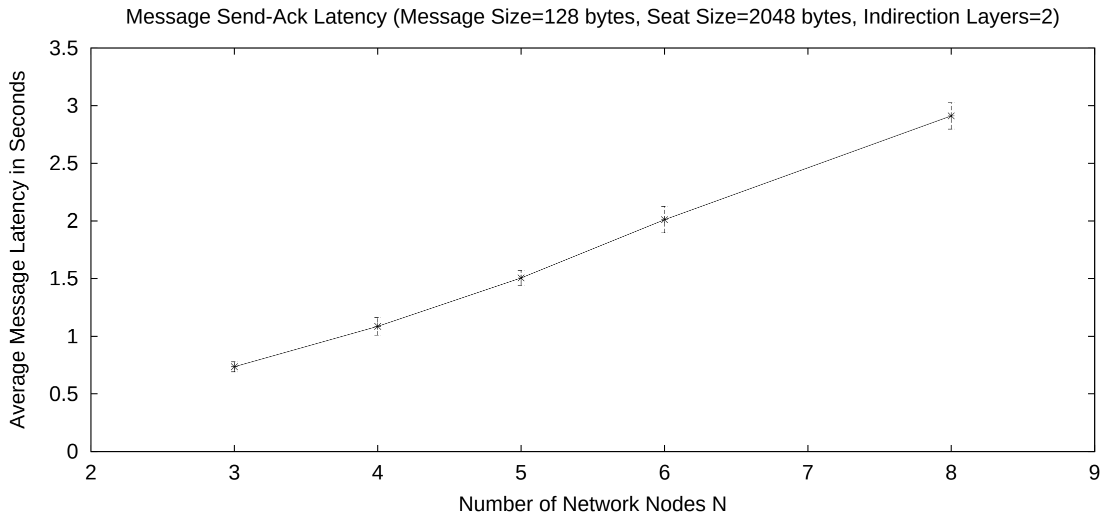

Figure 6.5: Effect Of Number Of Network Nodes In A Simple Traffic LAN Configuration

Figure 6.5 shows that the average message latency scales approximately linearly with the number of nodes in the network. This result makes sense since each additional node in the network increases message latency from two perspectives: the length of the bus tour increases, and the size of the Bus (number of seats) increases. This result is consistent with the algorithmic analysis of our protocol, and suggests that the protocol design is scalable to larger networks. The standard deviation of the message latency also tends to increase, because of the greater choice of indirection nodes available.

Figure 6.6 illustrates message latency results for a more general workload model in an 8-node Buses network. The average aggregate message arrival rate is ten 1024 byte messages per minute, with messages arriving according to a Poisson arrival process. The sources and destinations for each message are chosen uniformly at random. The experiment runs for just over half an hour, with 360 messages exchanged. A total of 360 messages were exchanged for this tests and all subsequent tests because of the larger variance in the results. The larger sample size increases the confidence of the statistical analysis. The results plotted in Figure 6.6 show the number of messages sent by each network node (the integer above each node ID number), as well as the mean and standard deviation of the message latency experienced by each node. Table 6.6 lists the parameters that were changed from the default parameter settings in Table 6.1.

Table 6.7: Relevant Parameter Settings For Figures 6.6 And 6.7

N	8
staticL	1
$uniformSeatSize$	2048 bytes

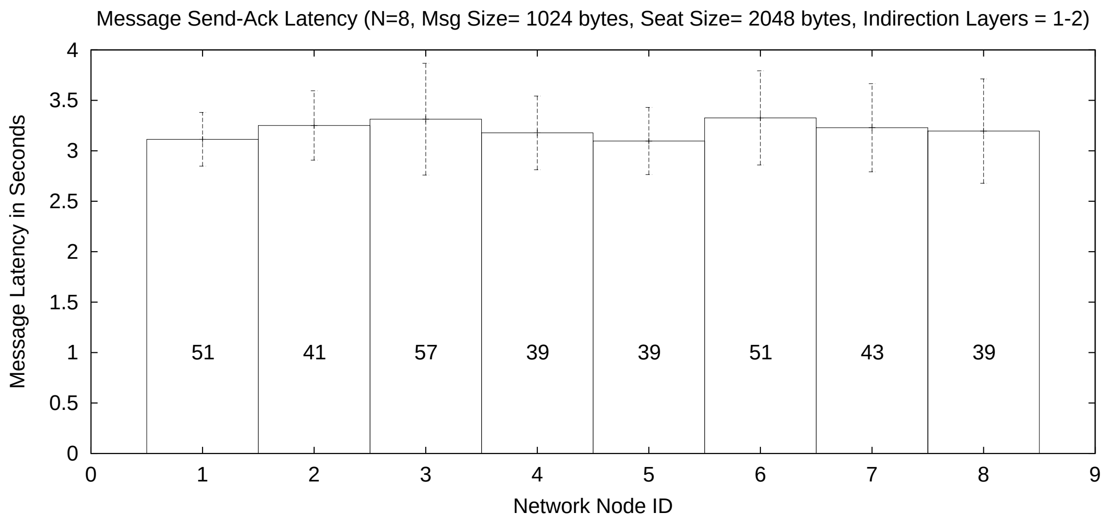

Figure 6.6: Effect Of Poisson Traffic In An Eight-Node LAN Configuration

Two observations are evident from Figure 6.6. First, the Practical Buses protocol is fair. Each network node experiences comparable latency for its messages, in terms of mean and standard deviation. Second, the mean message latency and especially the standard deviation of the message latency are higher for the random source-destination workload than for the static node1-node2 message workload for 8 nodes in Figure 6.5. This result makes sense because of the Poisson arrival process for messages, and the random sender-receiver pairs. These two factors create burstier traffic for each network node, which can lead to queuing delays for messages. Recall that the system has a threshold message arrival rate that it can sustain with steady performance (see Figure 6.4). However, a Poisson arrival process can occasionally cluster messages in time, causing transient queuing effects. Furthermore, the so-called “birthday paradox” for random sender-receiver pairs can produce a non-uniform distribution of messages in the network, again leading to queuing. These queuing delays account for the higher mean and standard deviation for message latency.

Figure 6.7 shows results for the Practical Buses protocol with two different RSA modules. For comparison, there is a slow RSA module and a fast RSA module that is 6.1 times faster. The purpose of the test is to show that RSA is the bottleneck. The two RSA modules were benchmarked by encrypting/decrypting 100 random 87 byte messages with newly generated keys for each encryption/decryption. The slow RSA module was an older version that did not use two standard RSA performance improvements: a small encryption exponent $e = 2^{16} + 1$ and the Chinese Remainder Theorem for decryption. The fast RSA module used these two improvements for RSA encryption/decryption primitives in both RSA-OAEP and RSA-SSA. For comparison, the exact same workloads are used for RSA and RSA 6X; the average aggregate message arrival rate is ten 1024 byte messages per minute, with messages arriving according to a Poisson arrival process. Also, the exact same parameter settings were used for RSA and RSA 6X, as in Table 6.8.

Table 6.8: Relevant Parameter Settings For Figures 6.7

N	8
$uniformSeatSize$	2048 bytes
$reduceDecrypt$	0

Figure 6.7 shows that RSA is indeed the bottleneck. RSA 6X reduced the average message latency by a factor of 7.54 and the standard deviation by a factor of 14.95. Average message latency was improved by more then a factor of 6.1 because in addition to improving the speed of decryption of all seats by a factor of 6.1 (RSA-OAEP), signature generation/verification was improved by a factor of 6.1 as well

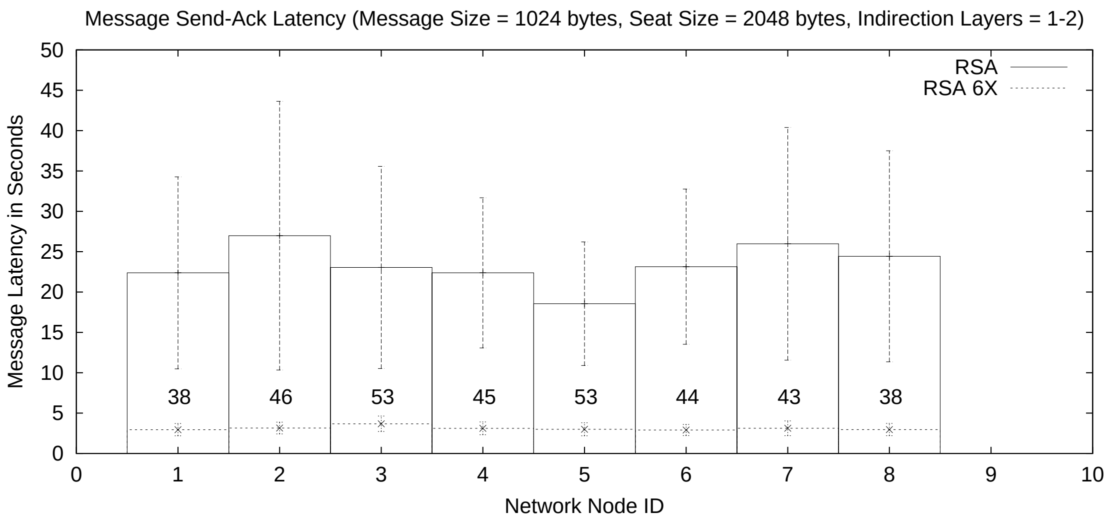

Figure 6.7: RSA Improvement And Effect In A Poisson Traffic Eight-Node LAN Configuration

(RSA-SSA). Note that signature verification is only checked on valid seats, not all seats because of the default $verifyAllSeats$ parameter setting of 0.

Even with the small e and Chinese Remainder Theorem improvements, RSA remains as the bottleneck of the protocol. The time it takes for a receiver to receive a message depends upon how fast the bus travels through the network. Figure 6.2 shows that the gigabit Ethernet network contributes a very small delay when the bus is sent from one node to the next. Therefore, the majority of the time it takes a bus to travel through the network is attributed to the bus delay at each individual node. Each node must process the bus, which requires decryption and verification of seats, the disguising of nested messages, the insertion of forward or acknowledgment messages into seats, the insertion of new nested messages into seats, and the generation of random data messages. Each of these actions requires at least one costly RSA encryption and/or a RSA decryption for each seat. Note that the bus does not wait for the generation of acknowledgments or new nested messages.

Lastly, Figure 6.8 illustrates the scalability of number of network nodes N with very general network conditions and a heavy traffic load. The mean message latency is plotted and the bars above and beneath represent one standard deviation. The average aggregate message arrival rate is 30 messages per minute with random sizes from 1 to 2048 bytes, arriving according to a Poisson arrival process. The individual mean for each node when $N \in [2, 4]$ is $30/N$ messages per minute. However, for $N \in [5, 14]$ two heavy traffic nodes (elephants) node_1 and $\text{node}_{\lfloor N/2+1 \rfloor}$ average 7.5 messages per minute each and the other nodes average $30/(2 * N)$ messages per minute. The number of indirection layers for each message is randomly chosen as either 1 or 2, to strengthen the anonymity of the protocol [17]. Each run takes 15 minutes, with 360 messages exchanged. Table 6.9 lists the parameters that were changed from the default parameter settings in Table 6.1.

Table 6.9: Relevant Parameter Settings For Figure 6.8

N	2,3,4,5,6,7,8,9,10,11,12,13,14
$uniformSeatSize$	3072 bytes
$maxOldSeat$	150 seconds
$maxDuplicateTime$	150 seconds
$ackTimeOut$	25 seconds
$maxResendTime$	100 seconds
$maxOldSeat$	150 seconds
R	2

The main observation from Figure 6.8 is that the average message latency scales almost linearly as N increases to 14. The message latency begins to slightly curve upwards in correlation to the increases in standard deviation. The linear increase

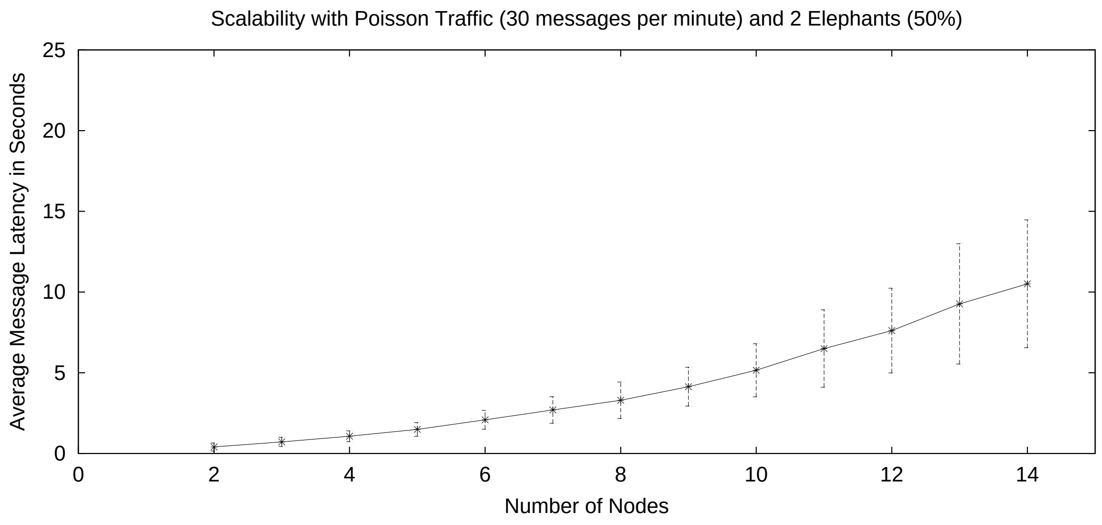

Figure 6.8: Effect Of Number Of Network Nodes In A General Traffic LAN Configuration

in average message latency as N increases is attributed to the linear increase in the number of hops the bus must do and the number of seats on the bus. However, the system's increased sensitivity to bursty traffic as N increases causes the curving of average message latency. As the number of nodes increases, the time it takes for a bus to complete one tour of the network increases. As a result, queues are emptied slower and the Buses network is more sensitive to bursts in traffic. Also, this increased sensitivity to bursts as N increases results in the standard deviation of the message latency taking a sudden jump when $N = 13$. This is confirmed by comparing Figures 6.9 and 6.10, which contain a node-by-node analysis for the two cases in Figure 6.8 where $N = 12$ and $N = 13$ respectively. For Figure 6.10, the elephant node$_1$ experiences a higher average and much higher standard deviation of message latency than in Figure 6.9. Also, node$_2$, node$_6$, node$_{12}$, and node$_{13}$ experience a higher average and standard deviation of message latency. The reason why the other nodes, including the elephant node$_7$, do not experience a significant degradation in performance is two-fold. First, when choosing their random indirection path, they do not frequently choose a node in their indirection path that is currently experiencing a significant burst of traffic. Second, they do not frequently experience significant bursts of traffic when they are trying to send a message. Note that traffic is not only created by sending a message, but also by the acknowledgment created in response to a received message.

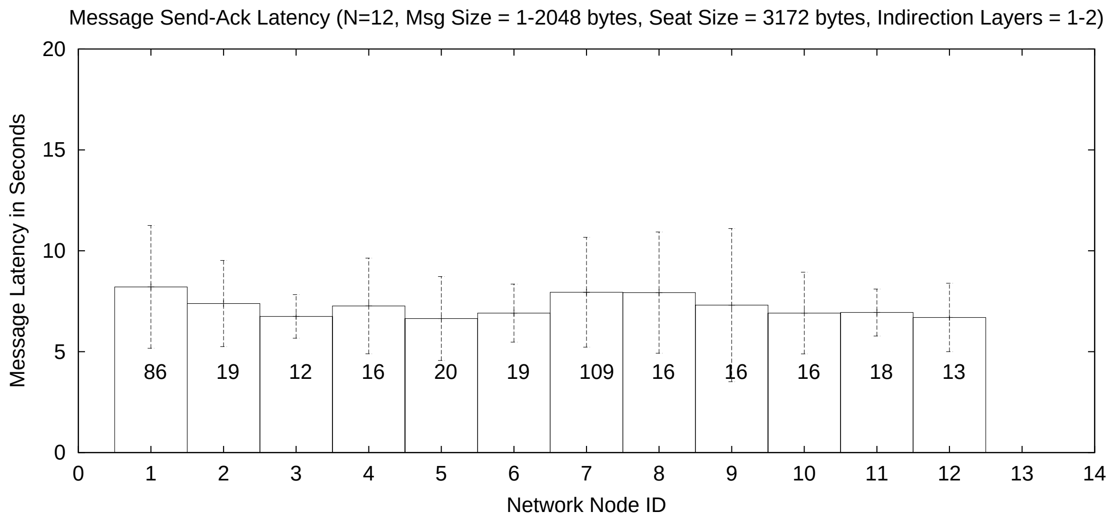

Figure 6.9: 12 Network Nodes In A General Traffic LAN Configuration

Two additional observations can be made from Figure 6.8 (Poisson traffic with two elephants) when comparing the results to Figure 6.6 (Poisson traffic). First the average message latency increases. Second, the standard deviation increases. Both of these observations agree with a theoretical analysis. The heavier traffic load of 30 messages per minute would increase the probability of nodes experiencing larger bursts of traffic. Also, bursts are increased because two nodes are each generating 1/4 of the message traffic instead of 1/8. The chances of these two nodes experiencing

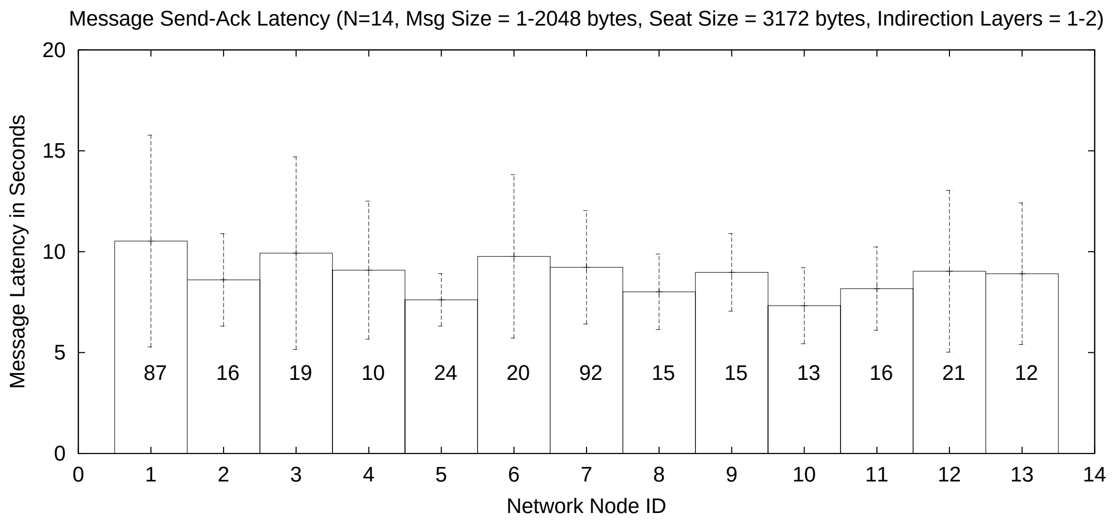

Figure 6.10: 13 Network Nodes In A General Traffic LAN Configuration

larger bursts of traffic are significantly increased. As a result, any messages that are sent, received, or indirected through the two heavy traffic nodes are more likely to experience queuing delays, increasing both the average and standard deviation of the message latency.

6.1.4 Summary

The experiments demonstrate the functionality of the Practical Buses protocol, and highlight its performance characteristics. The results show that message latency is largely independent of message size, but linearly dependent on seat size, the number of network nodes, and the number of indirection layers used. However, the linear dependence on seat size does not contribute significantly to the message latency. If the aggregate message arrival rate exceeds the threshold of messages that can be handled without queuing delay, the message latency increases sharply. However, if the high message arrival rate is not sustained, the queues will eventually be emptied and message latency will revert back to normal. In the last test, Figure 6.8, at least 30 aggregate messages per minute with random size 1 to 2048 bytes could be sustained in a network of size 2 to 14 nodes. The protocol performs reasonably in a moderately-sized network with the bus circulating in a round-robin tour. The bottleneck of the protocol is RSA encryption and decryption; furthermore, improvements to RSA directly improve the performance of the Practical Buses protocol as shown in Figure 6.7.

6.2 Anonymity Analysis

In this Section, the anonymity of the Practical Buses protocol is evaluated. Due to the changes to the protocol, all surveyed attacks are re-evaluated in Section 6.2.1. Then, Section 6.2.2 contains an analysis of the Practical Buses protocol for new attacks. A possible solution for each new vulnerability discovered is presented.

6.2.1 Resistance to Previous Attacks

The design of the Practical Buses protocol differs from the optimal buffer complexity Buses protocol in [4]. As a result, it is necessary to re-evaluate its susceptibility to all of the surveyed attacks.

Eavesdropper Attacks

The Practical Buses protocol is not vulnerable to any of the eavesdropper attacks listed in Section 3.2. The reasons follow:

- Local eavesdropper before proxy attack — Let proxy denote the first node after the sender. The local eavesdropper cannot identify a sender when it places a valid seat on the bus, because it is not known how to distinguish between random data and hybrid RSA-OAEP/CBC-AES nested encryption. Ciphertexts produced by RSA-OAEP are statistically random [21] and ciphertexts produced by the AES block cipher are statistically random [39]. The best the local eavesdropper can do is to monitor the replacement of seats, specifically multiple bus tours where a seat is not modified. However, the local eavesdropper cannot determine if the seat not being modified is a seat being forwarded or a seat the sender is sending according to the p-threshold replacement back-off scheme.

- Local eavesdropper after proxy attack — Let proxy denote the last node in the indirection path before the receiver. The local eavesdropper cannot identify a receiver because it is not known how to distinguish between random data and a single hybrid RSA-OAEP/CBC-AES encryption. RSA-OAEP is believed to be semantically secure [21] and AES ciphertexts are statistically random [39].

- Size correlation attack — A global eavesdropper cannot trace a message through the network based upon size, because all seats have uniform size.

- Time correlation attack — A global eavesdropper cannot trace a message with a time correlation attack, because a node decrypts every seat on the bus with its RSA-OAEP private key every time it receives the bus. The reception of a valid seat introduces the extra timing delay of AES decryption. However, AES decryptions introduce a very small timing delay and it would be infeasible for an adversary to determine if the timing delay was the result of an AES decryption. For example, time slicing among the bus server's threads and other processes could cause noticeable delays. Also, the Bus thread only waits for the Read thread to have received the bus before forwarding its modified bus. Hence, the Read thread decrypts a variable number of seats which further confuses the source of any noticeable delays.

- Low load attack — A global eavesdropper cannot identify the receiver in a low load network. The nodes observed deleting seats due to a message being sent and its corresponding acknowledgment are {message's indirection path to the receiver, receiver, acknowledgment's indirection path to sender, and sender}. However, randomly delayed seat deletion ensures that the global eavesdropper can only reduce the receiver to being in the set {indirection nodes to receiver, receiver, indirection nodes to sender, and sender}. In a low load network, the receiver would have seats immediately ready to send the acknowledgment and the global eavesdropper cannot determine where the message indirection ends and where the acknowledgment indirection begins. Also, a global eavesdropper cannot identify a sender in a low load network because it can't distinguish between a hybrid RSA-OAEP/CBC-AES encryption and random data. The p-threshold replacement back-off scheme also hides when a sender sends/resends a message. The global eavesdropper can monitor the replacement of seats, specifically n bus tours where a seat is not modified. The probability of a seat not being replaced after n times is $(1 - p^n)$. Hence as n increases, the probability of the node being a sender increases. However, this attack can be eliminated by setting the threshold r (the number of bus tours before a seat containing the sender's nested message must be disguised) to 0 and eliminating the back-off of r in the p-threshold replacement back-off scheme. In addition, the global eavesdropper can monitor seat deletions of the acknowledgment to determine who the sender is. However, randomly delayed seat deletion only lets the global eavesdropper identify the sender as being in the set {indirection nodes to receiver, receiver, indirection nodes to sender, and sender}.

- Marker attack — Since nested encryption is used, a global eavesdropper cannot trace the nested message through the network because of the layered encryption. Also, the seats that contain a nested message before and afterwards do not contain a common marker.

- Intersection attack — A global eavesdropper can only identify the receiver and sender belonging to the set {indirection nodes to receiver, receiver, indirection nodes to sender, sender}. However, it cannot correlate which of the sets belong to the same receiver and sender. The intersection of a subset of these sets may yield a common indirection node. With each new intersection, there is a reduced probability of correctly identifying the sender or receiver because each intersection increases the probability of incorrectly identifying an indirection node as the sender or receiver. A further complication is that the global eavesdropper does not know if it obtains multiple sets for the same party communicating or different parties communicating. In addition, unless it is a low load network, the global eavesdropper is not able to determine which set of deletions belong to the same message being sent and acknowledged. Thus the intersection attack does not provide the global eavesdropper with any additional information to reduce the sender or receiver to a smaller set with significant confidence.

Passive Adversary Attacks

The Practical Buses protocol is not vulnerable to any of the passive adversary attacks listed in Section 3.2. The reasons follow:

- Full compromised path attack — A passive adversary that knows the entire indirection path can identify the predecessor of the first hop in the indirection path as the sender. However, for a passive adversary to conclude that it controls the entire indirection path requires the adversary to corrupt the sender. Otherwise, what the adversary thinks is the sender could be forwarding a message. Clearly, it must be assumed that the sender is not corrupted when we analyze sender anonymity. Similarly, receiver anonymity is preserved.

- Timing attack — A passive adversary cannot launch a timing attack, because the requests for embedded Web objects do not happen immediately, but on the next bus tour. The passive adversary cannot tell who is the sender if the adversary is the first hop in the indirection path because of the p-threshold replacement back-off scheme hiding insertions of new messages. In addition, each time a nested message is disguised a new indirection path is used, reducing the probability that the passive adversary still controls the first hop in the indirection path.

- Passive traceback attack — Since there are no routing tables for the indirection paths, an adversary cannot corrupt routing tables on the reversed indirection path to identify the sender.

Active Adversary Attacks

The Practical Buses protocol is not vulnerable to any of the active adversary attacks listed in Section 3.2. The reasons follow:

- Replay attack — The Practical Buses protocol incorporates replay protection. $maxDuplicateTime$ must be set high enough to prevent an adversary from replaying a message. If the adversary is patient enough to wait for the maxDuplicateTime to lapse for each replay, it can only identify the possible set that a sender and receiver belongs to by observing the randomly delayed seat deletions. Further complicating the problem, the adversary cannot determine which seat deletions belong to which send-ack pair.

- Spam attack — The Practical Buses protocol does not use batching, and an active adversary does not reduce the obfuscation of inputs and outputs of a node by launching a spam attack.

- Mob attack — The user can reduce their anonymity to improve performance with significant effort (re-setting of parameters and recompilation of code), but the user is able to regain its anonymity at another time. To regain its anonymity the user resets its parameters, recompiles its code, and using a new alias and public/private key pair.

- Filtering attack — A filtering attack on the Practical Buses protocol would allow an adversary to filter the bus to one seat of interest before forwarding it. However, the active adversary does not gain any new knowledge from this attack. If it is a valid seat, then the active adversary sees the base case of a replay attack (the nested message is only sent once) and only observes the randomly delayed seat deletions. Otherwise, the seat contains random data and there are no resulting observable events that give the active adversary additional information.

6.2.2 Vulnerabilities and Solutions

It is necessary to analyze the Practical Buses protocol with respect to new attacks. The approach to identify these new attacks involves analyzing the protocol from the perspective of an eavesdropper, passive adversary, and active adversary. The observable events are listed for each threat model, new attacks are derived based on analysis of the observable events, and solutions to the attacks are presented. Since the original objective is to design a Practical Buses protocol that is resistant to all attacks in the literature, the solutions to these new attacks are not incorporated into the prototype and are left as future work. Recall, it is assumed that an attacker cannot tell the difference between a hybrid encryption RSA-OAEP/CBC-AES and random data.

Great care was taken when trying to discover all the potential attacks, but it is next to impossible to prove that a scheme is completely secure. Even if all the possible scenarios can be enumerated and each scenario is proven to be secure to all possible attacks, this does not prove that the system is secure. Assumptions may be incorrect, one or more possible scenarios may have been missed, or one or more attacks may have been missed. For example, RSA and AES have not been proven to be secure. It can only be proven that they are not vulnerable to any known attacks and the trust in their security increases as they withstand scrutiny over time.

New Global Eavesdropper Attacks and Solutions

There are only two observable events, and each allows a global eavesdropper to launch an attack. In low load networks, a global eavesdropper can identify a superset of the set {sender, receiver} by observing the randomly delayed seat deletions. These randomly delayed seat deletions result from the message being sent and the ensuing acknowledgment. The observable set of nodes that delete seats for a single send-ack are {indirection nodes to receiver, receiver, indirection nodes to sender, sender}. This attack reduces the possible set of nodes that a receiver and sender belong to, but preserves mutual anonymity. Let this attack be denoted *deletion observation attack*. Also, an adversary could observe a sender that sends a nested encrypted message multiple times without it being disguised, increasing the probability that the node is a sender but preserving unlinkability. Let this attack be denoted *resend observation attack*.

Preventing a deletion observation attack is simple. The protocol can be modified not to use randomly delayed seat deletion at the sender and receiver. Instead, when a node receives an application or acknowledgment nested message where the forward field is false, then it sends an *anonymous seat deletion request* to another node via a nested encrypted message. Let us denote this technique as *hidden end-points*. The difference of handling an anonymous seat deletion request is that the receiver of an anonymous seat deletion request does not send an acknowledgment, the deletion of the seat is an implicit acknowledgment for the sender of the anonymous seat deletion request. With hidden end-points, the observable set of nodes that delete seats are {indirection nodes to receiver, indirection nodes for receiver's anonymous seat deletion request, receiver of anonymous seat deletion request, indirection nodes to sender, indirection nodes for sender's anonymous seat deletion request, receiver of anonymous deletion request}. Hence, hidden end-points does not allow an adversary to utilize the observations of seat deletions to reduce the set that the sender and receiver belong to.

Preventing a resend observation attack is also simple. The observation can be eliminated by setting the p-threshold replacement back-off scheme's threshold and back-off to zero. The protocol already has the sender keep track of all disguised messages sent in the *sentList* via their seat tags. Thus the only additional work is the sender having to disguise the message every bus tour until the first hop in the indirection path deletes one of the variations or an anonymous acknowledgment from the receiver is received.

New Passive Adversary Attacks and Solutions

A passive adversary can observe the event when a nested message destined for a corrupted node is received because the decrypted nested message has the correct redundancy (a valid seat). Thus, when receiving a valid seat, a passive adversary can identify the predecessor in the indirection path (the owner field of the seat) as well as the ensuing embedded nested object that it forwards. In conjunction with a deletion observation attack in a low load network, each node that the passive adversary controls in the indirection path can be removed from the set of observed seat deletions. However, hidden end-points can eliminate this attack's reduction in anonymity. Hence, receiver anonymity is preserved. The most extreme attack on sender anonymity is in a low load network where the corrupted node is the first hop in the indirection path. If no seat deletions occur within an acceptable time window before this point, as soon as the adversary receives a valid seat it can immediately identify the sender as the owner of the valid seat. Let the attack be denoted as *first send attack*.

To protect against a first send attack, a node should keep track of randomly delayed seat deletions and record the time since the last observed seat deletion. If no seat deletions are observed past a certain low load time threshold, a node is forced to send a dummy nested message. Each node can individually turn these dummy nested messages on or off depending on whether they want protection against the first send attack and preserve mutual anonymity or if they want to potentially improve performance at the cost of unlinkability. Note that one node sending dummy messages is sufficient to provide enough deletions; no other node would have to create constant traffic. Even if all the nodes create dummy messages in the absence of seat deletions beyond an acceptable time threshold, the one with the lowest time threshold that first receives the bus would create sufficient dummy messages resulting in deletions and prevent the other nodes from creating dummy messages. Thus, on average one out of N nodes would create a dummy message for each bus tour and as N increases the dummy message overhead approaches zero, preserving scalability. Note that more than one node could create a dummy message if multiple nodes reached the low load time threshold on the same bus tour, and each sent a dummy message which would not result in seat deletions until at least the next bus tour. Let us denote the scenario where exactly one node sends a dummy message per bus tour as *confusion traffic*.

New Active Adversary Attacks and Solutions

An active adversary has the following additional abilities above and beyond a passive adversary; it can delete, modify, or add messages. This allows it to create new observable events. The ability to delete seats allows an active adversary to perform a DoS attack, or to delete seats which may cause a sender to resend a message. Let the former be called *DoS seat deletion attack*, and the latter be called a *forced resend attack*. The ability to modify seats only allows an adversary to change the seat fields it owns and create a new signature. The active adversary cannot perform modifications to a valid nested message without corrupting the OAEP redundancy nor can it make modifications to a seat it does not own without corrupting the digital signature. Let us denote the attack where seat fields are changed as *seat field modification attack*. Lastly, an active adversary can add messages. Since the active adversary is the sender it can attack receiver anonymity by spamming the receiver with multiple valid messages. Let the attack be denoted as *spam receiver attack*. Also, the adversary could insert more than k messages destined for a target node, to overload the node with nested messages to forward so that it cannot send any messages of its own. Let the attack be denoted as a *overload DoS attack*.

The *forced resend attack* can cause a sender to resend the message with multiple indirection paths. An active adversary can hope to identify the sender by observing the sender resending a message multiple times with different nested encryptions. However, the adversary is required to be consistently on the indirection path to delete a received valid seat. Otherwise, the adversary must find a 'needle in a hay-stack' by randomly deleting at least one seat in each random indirection path. Hence, we assume that the adversary is in the indirection path of the first nested message sent. However, an active adversary cannot guarantee it will be in the indirection path since the sender chooses a new random indirection path each time. If the threshold (number of resends for a particular nested message) and back-off are too high, then a corrupted node on the bus tour can statistically increase the probability of identifying the sender by observing it resend the same message multiple times. Thus, the chance of a forced resend attack compromising sender anonymity is very low. The probability that the node is identified as a sender $1 - p^{\sum_{i=1}^{n} \text{resends}_{m_i}}$ where resends_{m_i} is the number of times the i^{th} nested message for the same message is observed being re-sent according to the p-threshold replacement back-off scheme and n is the total number of observed disguised messages sent. Assuming that the 'needle in a hay-stack' analogy restricts the attacker to being on the first indirection path and hoping that it is on the next indirection path results in the probability of a node being identified a sender being reduced to $1 - p^{\sum_{i=1}^{n} \text{resends}_{m_i} (l/N)^{i-1}}$ where l is the average number of indirection layers and N is the number of nodes participating in the Practical Buses protocol. However, to defend completely against the attack the sender can either reduce the threshold and back-off to zero in the p-threshold replacement back-off scheme or reduce the number of times a message can be resent due to the absence of an acknowledgment.

The *DoS seat deletion attack* can cause a DoS attack for a particular node if the active adversary can delete all the messages a node sends. However, a corrupted node on the bus tour deleting all received valid seats would not suffice because it cannot guarantee to be on all of the indirection paths. Nor would a corrupted node deleting random seats on the bus tour suffice because of the 'needle in a hay-stack' problem. This implies that the active adversary has to be between the sender and receiver and be able to delete the sender's seat or overcome the 'needle in a hay-stack' analogy and delete any of the indirection seats before they are received. Deleting any of the indirection seats requires the active adversary to analyze each seat on the bus and try to trace the reversed indirection back to the sender. This would require the adversary to record each bus transformation and iteratively correlate a nested message object to the nested message object that contained it. However, without possessing the private keys of all the outer layers of a nested message object or breaking the encryption scheme an active adversary cannot trace a seat containing a nested message back to the sender. Thus, an adversary must be the node that the sender forwards the bus to, or it can randomly delete seats hoping it deletes an indirection seat before it is received. The probability of success of the latter would be very small if enough resends occur but the former would be successful.

A sender cannot prevent a DoS attack but it could determine where the attack is taking place. For example, the sender could send a *trace application* nested message that would cause each node in the indirection path to send an anonymous acknowledgment to the sender. The sender then sends the trace application nested message and knows that the message is lost between two indirection nodes. It can then monitor the network between these two indirection nodes to determine where the message is lost because it knows what each layer of the nested encrypted message should look like.

The *seat field modification attack* poses the threat that if an active adversary receives a valid seat, it can continue to replay the seat by changing the timestamp field, attacking receiver and sender anonymity by observing the set of nodes performing seat deletion. Changing the owner field does not allow an attack; it only improves sender anonymity by making it more difficult to trace the indirection path back to the sender. Changing the tag field only prevents the seat from being deleted by the next hop in the indirection path, resulting in resources being tied up for the active adversary. The ability to launch a replay attack with the timestamp field would only result in the observable events of randomly delayed seat deletion. Also, the replays would have to to spaced sufficiently apart to prevent the replay protection from ignoring the replayed seats. However, hidden end-points would prevent the attack from identifying the receiver or sender. Hence, the *seat field modification attack* only poses a threat in a low load network where hidden end-points are not used.

The *spam receiver attack* consists of an active adversary sending multiple different messages encrypted with the receiver's anonymous public key in order to identify the receiver and bind the anonymous public key to the receiver. To identify the receiver, the best the active adversary can do is observe the randomly delayed seat deletions.

If the active adversary does this with an indirection path of length zero in a low load network, eventually the intersection of all these sets will result in a set containing only the receiver and the already known sender since the resulting indirection path for the acknowledgment is random. To prevent this attack, the receiver can use hidden end-points and the active adversary can no longer observe the receiver deleting a seat.

The *overload DoS attack* consists of an active adversary sending multiple nested messages destined for the same node, so that it is overloaded with nested messages to forward. Since the $forwardAckQueue$ has priority over the *sendQueue*, the adversary will not be able to send any messages. As a side effect, this would decrease k and reduce the anonymity of the adversary as a sender since it violates the conditions of the p-threshold replacement back-off scheme. To prevent this attack, any node that receives a bus ensures that no node owns more than the allowed upper bound of K seats. If a node owns more that K seats, it can randomly delete seats owned by the violating node until the node owns K seats. Let us denote this counter-measure as *prune max seats.*

6.2.3 Summary

The new attacks reduce anonymity for the sender or receiver, and in the most extreme case they identify the sender or receiver but preserve unlinkability. However, new techniques were introduced to defend against these attacks and preserve mutual anonymity. Table 6.10 lists each new attack, its impact on mutual anonymity, and the counter-measures used to defend against the attack.

From the analysis of Table 6.10, it is apparent that setting the threshold and

Table 6.10: New Attacks Against The Practical Buses Protocol And Solutions

Attack	**Threat Model**	**Impact On Mut. Anonymity**	**Counter Measure**
deletion observation	global evdpr.	mutual anonymity: reduced set for sender/receiver	confusion traffic or hidden end-points
resend observation	global evdpr.	mutual anonymity: prob. sender ident. +	confusion traffic or threshold,back-off=0
first send	pass. advsry.	identify sender, unlinkability	confusion traffic
forced resend	actv. advsry.	mutual anonymity: prob. sender ident. +	limit resends or threshold,back-off=0
DoS seat deletion	actv. advsry.	none, DoS	trace application nested message and monitor suspect nodes
seat field modification	actv. advsry.	mutual anonymity: reduced set for sender/receiver	confusion traffic or hidden end-points
spam receiver	actv. advsry.	unlinkability identify receiver	hidden end-points
overload DoS attack	actv. advsry.	none, DoS	Prune Max Seats

back-off to zero, using confusion traffic, using hidden end-points, and prune max seats will protect against all of these attacks. In addition, all of these improvements are scalable. However, not all of these improvements are necessary. The improvements needed depend upon the requirements of the application: what type of anonymity is to be provided and for what level of threat model the protocol should provide anonymity. If none of these improvements are made, unlinkability is preserved and the only attack that can completely identify a sender is first send attack and the only attack that can completely identify a receiver is a spam receiver attack.

The caveat of the Practical Buses protocol is that mutual anonymity is provided from all perspectives only if the public key used to encrypt the inner-most core of a nested message is not correlated to the receiver. For example, the public key that identifies a node's seats is not suitable to encrypt the inner-most core. The sender can correlate the public key to the node that modifies the seats and defeat receiver anonymity. Instead, the anonymous public key that is used in the nested message for an anonymous acknowledgment would preserve mutual anonymity. However, the sender cannot request this anonymous public key using the Practical Buses protocol, otherwise the receiver binds the anonymous public key with receiver's public key used to send the anonymous public key request. Another means, such as a Web posting or anonymous certificate, is required to bind the anonymous public key to an anonymous identity. For example, a psychologist could have an anonymous certificate identifying it as a psychologist, and this certificate could be signed by a trusted third party to vouch that the owner of the public key is indeed a psychologist. However, using the public key to identify seats suffices if a sender-receiver pair requires mutual anonymity from the perspective of any other node.

Chapter 7

Conclusions and Future Work

This chapter concludes the thesis. In Section 7.1, the work that has been done is summarized. In Section 7.2, the main conclusions are presented. Lastly, in Section 7.3 future directions for research are discussed.

7.1 Thesis Summary

Anonymous communication is required so people can communicate without fear of retribution. There are numerous applications that need anonymous communication. Lack of anonymity may suppress communication, resulting in psychological, social, and financial losses, or even the loss of life.

A robust anonymous communication scheme should meet many design criteria: security against all attacks in the literature, scalability, mutual anonymity, and support for a dynamic topology. However, none of the prior anonymous communication schemes in the literature support all the criteria. A literature survey of anonymous communication schemes and analysis showed that all schemes except DC-Net are vulnerable to at least one attack. However, DC-Net is not scalable and Chaum's Mixes or Buses could not support all the criteria even if they were made secure by incorporating the security characteristics as outlined in Chapter 3.

The Practical Buses protocol is a robust solution that supports all the design criteria. Buses was chosen as the best candidate and re-designed into the Practical

Buses Protocol. Three new techniques are introduced in the core of the design: nested encryption with indirection, p-threshold replacement back-off scheme, and randomly delayed seat deletion. The new design enhances security and anonymity, and improves fault tolerance and performance. Security and anonymity were enhanced with the addition of anonymous acknowledgments and signed bus seats. Fault tolerance is augmented with resends and acknowledgments handling lost seats and bus loss tolerance handling lost buses. Performance was improved by hybrid encryption, dynamic bus seats, deletion of expired bus seats, and reduced seat decryptions.

To show that the Practical Buses protocol is scalable, the prototype presented in was implemented. The prototype was experimentally evaluated and it was shown that the Practical Buses protocol has good performance and scales well for a static topology. In addition, the anonymity provided by the Practical Buses protocol was carefully analyzed. This analysis showed that the Practical Buses protocol is secure against all previous attacks. New techniques to defend against a set of newly discovered attacks are discussed.

7.2 Conclusions

There are three main conclusions from this thesis:

1. The prior analysis of anonymous communication schemes was not sufficient to provide complete anonymity. 36 new attack vulnerabilities were discovered for the thirteen anonymous communication schemes analyzed in Table 3.1 and 3.2. In many cases the analysis of these schemes is done by the inventor, the threat model is limited, and numerous attacks that exist in the literature are missed.

A more robust anonymity analysis is presented in this thesis, summarizing all the attacks in the literature, identifying the characteristics required to defend against each attack, and identifying the characteristics each scheme used. In addition, the characteristics are identified to protect against all surveyed attacks for anonymous communication schemes based upon mixes, buses, or broadcasting.

2. The Practical Buses protocol is a promising anonymous communication scheme. It is the first anonymous communication scheme in the literature to meet robust design criteria: provide security against all attacks in the literature, scalability, mutual anonymity, and support for a dynamic topology. All prior anonymous schemes including the original Buses protocol do not meet all the design criteria.

3. The Practical Buses prototype is a practical solution with real world uses. Although our prototype can handle a dynamic network topology, the random walk bus traversal is too inefficient to provide reasonable scabalility. Thus, our prototype is restricted to static networks. Despite this restriction, there are numerous real world applications that could have beneficial results. A static dedicated Practical Buses network could be set-up for group employee evaluations, group therapy sessions, military communications, scheduled employee feedback sessions, et cetera. Furthermore, a static dedicated Practical Buses network could be extended to support additional applications such as anonymous Web browsing, e-mail, e-voting, e-counseling for victims of abuse, et cetera. For Internet deployment the prototype would have to be modified

so that each node securely stores its own private key, the central static routing table would have to be updated for the static topology, the central public key table would have to be updated to include all the nodes public keys, and the central alias-public key mapping would have to updated. In addition, application support would have to be added. The sender would send an anonymous request for a random node in the Practical Buses Network to send the message on behalf of the sender and the random node would parse the request, send the request, and include the reply anonymously within the anonymous acknowledgment.

7.3 Future Work

The Practical Buses protocol could be extended to improve generality, functionality, performance, scalability, and anonymity. Each improvement could be achieved by re-designing, implementing, measuring, and evaluating each improvement to the Practical Buses Protocol.

Generality

To support general networks, the Practical Buses Protocol should efficiently support a dynamic network topology such as an ad hoc wireless network. The topology of the network changes rapidly and paths could disappear before an anonymous message travels its entire route. A random walk would allow anonymous communication in a dynamic network topology, but would not scale well since the expected number of hops would be $O(mn)$ per bus tour where m is the number of edges and n is the number of nodes. One potential solution that uses random walks is to divide the network into clusters, have a bus for each cluster, and have seat transfers between clusters. Using this method, the expensive tail of $O(mn)$ could be cut off before it grows too quickly but seat transfer, shortest path information, and cluster management would be needed. Another approach is to create dynamically a static path whenever it is determined that the network topology has changed. However, the efficiency of the solution would depend on the overhead of creating a static path dynamically being compounded by how frequent the network topology changes.

Functionality

The Practical Buses protocol's functionality could be extended to provide message fragmentation, application transparency, and DoS counter attack measures. A malicious node (e.g., a node performing a DoS attack) could be detected by a Practical Buses Protocol layer traceback. Each node in the indirection path would be required to send an anonymous acknowledgment back to the sender. Then, the sender could monitor the network between the two indirection nodes where the message is lost to identify the malicious node, since it knows what each layer of the nested message looks like and can verify each seats signature. In addition, the Practical Buses protocol set-up should be decentralized so as not to require central administrative set-up.

The functionality should also be extended to include controlled anonymity revocation, otherwise the Practical Buses protocol could be used for illicit purposes. Anonymity revocation could require all nodes to only use public/private key pairs provided by a trusted third party. This trusted third party could then try all possible private keys on a seat that is having its anonymity revoked, to remove each layer of indirection. To prevent abuse by a trusted individual, pieces of the private key could be secretly distributed to a group of individuals so that a certain threshold is required to reconstruct the private key. However, anonymity revocation contradicts decentralization and both goals cannot be achieved together. Anonymity revocation requires centralized trusted parties.

Performance

Improvements to the protocol's performance are also possible. For example, incomparable public keys [46] could obviate the need for a receiver to decrypt all bus seats with both its private key and anonymous private key. Instead, a single private key could be used decrypt ciphertexts that are produced with its public and anonymous public keys. Also, RSA-OAEP remains the performance bottleneck in our protocol. Minimizing the number of RSA-OAEP encryptions and decryptions required would improve the protocol performance. In addition, improving the speed of the RSA-OAEP encryption/decryption module or replacing it with a faster cryptosystem such as elliptic curve cryptography would improve the protocol performance. Also, performance can be improved by exploiting the parallelism available in our protocol implementation. Lastly, additional performance testing and parameter tuning would improve performance.

Scalability

Improvements are needed to improve the scalability of the protocol by reducing the average message latency. Potential scalability improvements are clustering, multiple buses on a route, and traffic shaping. Clustering could limit the size of each bus route to a manageable size. Multiple buses on a route would decrease the time a node has to wait for a bus to insert a seat. The increased number of buses would also allow the node to send more messages on the larger amount of seats. However, this would increase the communication and computational complexities of the protocol. The burstiness of the traffic could be reduced by using leaky buckets at the traffic sources. Leaky buckets are used in ATM for constant bit rate and variable bit rate to shape the traffic.

Anonymity

To defend against the new attacks in Chapter 6, the Practical Buses protocol should be modified. The threshold and back-off should be set to zero, confusion traffic should be added, hidden end-points should be added, and prune max seats should be added. With hidden end-points, the random seat deletion delay could be set to zero to improve performance, since the set of nodes observed deleting nodes would no longer include the sender or receiver. However, even with these changes the Practical Buses protocol could potentially have undiscovered vulnerabilities despite great care in designing a secure protocol. Potential weaknesses are missed code vulnerabilities such as buffer overflows, application vulnerabilities, operating system vulnerabilities, or viruses creating back doors to install software such as key stroke recorders.

Bibliography

[1] R. Aleliunas, R. Karp, R. Lipton, L. Lovasz, and C. Rackoff. Random walks, universal traversal sequences and the complexity of maze problems. In *Proceedings of the 11th Annual Symposium on the Theory of Computing*, pages 218–223. ACM, 1979.

[2] Anonymizer, 2003. http://www.anonymizer.com.

[3] Anonymous mailer, 2003. http://mailer.us.tf/.

[4] A. Beimel and S. Dolev. Buses for anonymous message delivery. *Journal of Cryptology*, 16(1):25–39, 2003.

[5] M. Bellare, J. Killian, and P. Rogaway. The security of cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–369, 1994.

[6] H. Berghel and K. Womack. Anonymizing the net. *Communications of the ACM*, 46(4):15–19, 2003.

[7] D. Chaum. Untraceable electronic mail, return addresses and digital pseudonyms. *Communications of the ACM*, 24(2):84–88, 1981.

[8] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1):65–75, 1988.

[9] J. Claessens, B. Preneel, and J. Vandewalle. Solutions for anonymous communication on the internet. In *Proceedings of the International Carnahan Conference*

on Security Technology, pages 298–303. IEEE, 1999.

[10] L. Cottrell. Mixmaster & remailer attacks, 2003. http://www.obscura.com /˜loki/re-mailer-essay.html.

[11] Crowds, 2002. http://www.research.att.com/˜crowds/.

[12] DC-Net, 2003. http://cypherpunks.venona.com/date/1992/12/msg00114.html.

[13] S. Dolev and R. Ostrovsky. Xor-trees for efficient anonymous multicast and reception. *ACM Transactions on Information and System Security*, 3(2):63–84, 2000.

[14] Freedom network, 2003. http://www.freedom.net.

[15] E. Gabber, P. Gibbons, and D. Kristol. On secure and pseudonymous client-relationships with multiple servers. *ACM Transactions on Information and System Security*, 2(4):391–415, 2000.

[16] GMP. http://www.swox.com/gmp/.

[17] Y. Guan, X. Fu, R. Bettati, and W. Zhoa. An optimal strategy for anonymous communication protocols. In *Proceedings of 22nd International Conference on Distributed Computing Systems*, pages 257–220. IEEE, 2002.

[18] C. Gulcu and G. Tsudik. Mixing e-mail with babel. In *Proceedings of the 1996 Symposium on Network and Distributed System Security*, pages 2–16. IEEE, 1996.

[19] A. Hirt, M. J. Jacobson Jr., and C. Williamson. Survey and Analysis of Anonymous Communication Schemes, 2003. Submitted to the ACM Computing Surveys.

[20] A. Hirt, M. J. Jacobson Jr., and C. Williamson. An Improved Buses Protocol for Anonymous Communication, 2004. Submitted to ACM Conference on Computer and Communications Security.

[21] RSA Laboratories. *RSA-OAEP Encryption Scheme — Algorithm Specification and Supporting Documentation.* RSA Security Inc, 2000. ftp://ftp.rsasecurity.com/pub/rsalabs/rsa_algorithm/rsa-oaep_spec.pdf.

[22] RSA Laboratories. *RSA Signature Scheme with Appendix — Probabilistic Signature Scheme*. RSA Security Inc, 2000. ftp://ftp.rsasecurity.com /pub/rsalabs/rsa_algorithm/nessie_pss.zip.

[23] B. Lewis and D. Berg. *Multithreaded Programming with Pthreads.* SUN Microsystems, Mountain View, California, 1998.

[24] Lucent personal web assistant, 2003. http://www.math.tau.ac.il /˜matias/lpwa.html.

[25] N. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[26] A. Menezes, P. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, Florida, 2001.

[27] National Institute of Standards and Technology. *Advanced Encryption Standard — FIPS 197*, 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[28] Onion routing, 2003. http://www.onion-router.net/.

[29] A. Pfitzmann and M. Waidner. Networks without user observability. *Computers & Security*, 2(6):158–166, 1987.

[30] Pipenet 1.1, 2003. http://www.eskimo.com/˜weidai/pipenet.txt.

[31] Proxymate, 2003. http://proxymate.com/.

[32] Pthreads. http://www.opengroup.org/austin/papers/backgrounder.html.

[33] Recent results on oaep security, 2004. http://www.rsasecurity.com /rsalabs/node.asp?id=2147.

[34] M. Reed, P. Syverson, and D. Goldschlag. Anonymous connections and onion routing. In *Proceedings of Symposium on Security and Privacy*, pages 44–54. IEEE, 1997.

[35] M. Reiter and A. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.

[36] SHA-1 implementation, 2000. http://www.faqs.org/rfcs/rfc3174.html.

[37] R. Sherwood, B. Bhattacharjee, and A. Srinivasan. p^5: A protocol for scalable anonymous communication. In *Proceedings of Symposium on Security and Privacy*, pages 53–65. IEEE, 2002.

[38] C. Shields and B. Levine. A protocol for anonymous communication over the internet. In *Proceedings of the Conference on Computer and Communication Security*, pages 33–42. ACM, 2000.

[39] J. Sotto and L. Bassaham. Randomness testing of the advanced encryption standard finalist candidates. Technical report, National Institute of Standards and Technologies, 200.

[40] S. Stefanek. AES implementation, 2000. http://www.mirrors.wiretapped.net /security/cryptography/algorithms/aes/aes-c++-sstefanek/rijndael.cpp.

[41] P. Syverson, D. Goldschlag, and M. Reed. Anonymous connections and onion routing. In *Proceedings of the Symposium on Security and Privacy*, pages 44–54. IEEE, 1997.

[42] P. Syverson, D. Goldschlag, and M. Reed. Onion routing access configuration. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, pages 34–40. IEEE, 2000.

[43] P. Syverson, D. Goldschlag, and M. Reed. *Designing Privacy Enhancing Technologies*. Springer-Verlag, Berlin, Heidelberg, 2001.

[44] Type 1 and 2 remailer list, 2003. http://freedom.gmsociety.org/stats/.

[45] Type 1 remailer list, 2003. http://anon.efga.org/Remailers/TypeIList/.

[46] B. Waters, E. Felten, and A. Sahai. Receiver anonymity via incomparable public keys. In *Proceedings of the 10th ACM conference on Computer and Communication Security*, pages 112–121. ACM, 2003.

[47] P. Zimmerman. PGP user's guide, 1994. included in PGP distribution 2.6i.

Appendix A

Anonymous Communication Scheme Examples

A.1 Mix Example: Chaum's Mixes

To further understand mixes, consider the following example based upon Chaum's mixes. Assume there are three intermediate mixes to send a message M and receive a reply M' and that the sender S sends the message M to the receiver R.

S begins by choosing a re-routing path, mix_1, mix_2, mix_3, and creates an untraceable return address so that R can reply. S creates three random pairs of public/private keys $(R_1^+/R_1^-, R_2^+/R_2^-, R_3^+/R_3^-)$ that also act as random salt. The untraceable address is created recursively using the sender's re-routing path to the receiver. A_S is recursively prepended with the randomly generated public key R_i^+, encrypted with the mixes public key K_i^+, and appended with the address A_i (for $i = 1, i = 2, i = 3$), that is:

$$A_S$$

$$E_{K_1^+}(R_1^+, A_S), A_1$$

$$E_{K_2^+}(R_2^+, E_{K_1^+}(R_1^+, A_S), A_1), A_2$$

$$E_{K_3^+}(R_3^+, E_{K_2^+}(R_2^+, E_{K_1^+}(R_1^+, A_S), A_1), A_2)$$

Note that for the final iterative loop, the address A_3 is not appended because R knows which mix sent the message M. However, after the last iterative loop K_S^+ is appended (a public key randomly generated for R's reply) so that R can encrypt its reply with K_S^+:

$$E_{K_3^+}(R_3^+, E_{K_2^+}(R_2^+, E_{K_1^+}(R_1^+, A_S), A_1), A_2), K_S^+ \tag{A.1}$$

Let us denote this untraceable return address in Equation A.1 as $return$.

Once the return address has been created, S prepares the anonymous message by layering the encryption of $(return, m)$. The layered encryption is done on the reverse path with R's public key, then mix_3's public key, then mix_2's public key, and finally mix_1's public key. Also, before encrypting each layer with K_i^+, the address of the next hop A_{i+1} is prepended to the message and random salt R_i is appended to the message. That is:

$$(return, m)$$

$$E_{K_R^+}(R_0, (return, m)) \tag{A.2}$$

$$E_{K_3^+}(R_3, E_{K_R^+}(R_0, (return, M)), A_R)$$

$$E_{K_2^+}(R_2, E_{K_3^+}(R_3, E_{K_R^+}(R_0, (return, M)), A_R), A_3)$$

$$E_{K_1^+}(R_1, E_{K_2^+}(R_2, E_{K_3^+}(R_3, E_{K_R^+}(R_0, (return, M)), A_R), A_3), A_2) \tag{A.3}$$

Note A_{R+1} is not appended in Equation A.2 because A_R is the last hop. Also, the address A_1 in Equation A.3 is not appended because S knows that the first hop is A_1.

S has prepared the encrypted layered message with the return address and must remember $R_1^-, R_2^-, R_3^-, K_s^-$ as the private keys to decrypt R's reply M'. Let $A \rightarrow B$

denote the message that A sends to B. Now let us trace the message as it is sent from S to R.

$$\begin{array}{ll}
S \rightarrow mix_1 & E_{K_1^+}(R_1, E_{K_2^+}(R_2, E_{K_3^+}(R_3, E_{K_R^+}(R_0, (return, M)), A_R), A_3), A_2) \\
mix_1 \rightarrow mix_2 & E_{K_2^+}(R_2, E_{K_3^+}(R_3, E_{K_R^+}(R_0, (return, M)), A_R), A_3) \\
mix_2 \rightarrow mix_3 & E_{K_3^+}(R_3, E_{K_R^+}(R_0, (return, M)), A_R) \\
mix_3 \rightarrow R & E_{K_R^+}(R_0, (return, M))
\end{array}$$

Note that the receiving mix_i of a layered message uses its private key K_i^- to decrypt the message, throws away the random salt R_i, and determines the next hop A_{i+1}. When R receives the message it decrypts the message with K_R^-, saves $(R_0, return)$ to reply, and reads the message M.

R then uses the untraceable return address to send a reply M'. Recall that K_S^+ is found in $return$ (see Equation A.1). R appends its reply M' with R_0 and encrypts (R_0, M') with K_S^+. This is appended with $E_{K_3^+}(R_3^+, E_{K_2^+}(R_2^+, E_{K_1^+}(R_1^+, A_S), A_1), A_2)$ from $return$ (see Equation A.1). Now, R sends $(E_{K_3^+}(R_3^+, E_{K_2^+}(R_2^+, E_{K_1^+}(R_1^+, A_S),$ $A_1), A_2), E_{K_S^+}(R_0, M')$ to mix_3 and the message arrives at S as follows:

$$\begin{array}{ll}
R \rightarrow mix_3 & (E_{K_3^+}(R_3^+, E_{K_2^+}(R_2^+, E_{K_1^+}(R_1^+, A_S), A_1), A_2), E_{K_S^+}(R_0, M') \\
mix_3 \rightarrow mix_2 & E_{K_2^+}(R_2^+, E_{K_1^+}(R_1^+, A_S), A_1), A_2), E_{R_3^+}(E_{K_S^+}(R_0, M')) \\
mix_2 \rightarrow mix_1 & E_{K_1^+}(R_1^+, A_S), A_1), A_2), E_{R_2^+}(E_{R_3^+}(E_{K_S^+}(R_0, M'))) \\
mix_1 \rightarrow S & E_{R_1^+}(E_{R_2^+}(E_{R_3^+}(E_{K_S^+}(R_0, M'))))
\end{array}$$

Note that the receiving mix$_i$ of a layered message uses its private key K_i^- to decrypt $return$, uses the public key R_i^+ to encrypt the reply message, and determines the next hop A_{i-1}. Once S receives $E_{R_1^+}(E_{R_2^+}(E_{R_3^+}(E_{K_S^+}(R_0, m'))))$ it uses the private keys $R_1^-, R_2^-, R_3^-, K_s^-$ to decrypt the message, uses the random salt R_0 to verify that R sent the reply message, and reads M'.

A.2 Re-routing Example: Crowds

To further understand how re-routing works, consider the following example based on Crowds. Let the sender (user web browsing) be denoted S and the receiver (web server) be denoted R. There are three phases in Crowds, authentication with the Blender, static-path creation, and anonymous web requests sends/receives.

S first contacts the Blender and authenticates itself with the username and password so it can become a member of a crowd. If the Blender authenticates S, it adds S's IP Address, port number, and account name to the list of crowd members.

S then waits for the next join commit, to create its static re-routing path. Upon receipt of the join commit, S's Jondoe forwards to a random Jondoe a randomly generated 128 bit path key and a randomly generated 128 bit path ID encrypted with its shared symmetric key. The IP, port, path ID of the first hop in the re-routing path as well as the path key are recorded by S's Jondoe so it knows where to send subsequent web requests tagged with that path ID. Each Jondoe that receives the re-routing path creation request flips a biased coin. With probability p_f the intermediate Jondoe forwards to a random Jondoe a new random 128-bit path ID and the same path key encrypted with the random Jondoe and intermediate Jondoe's shared symmetric key. In addition, the old and new path ID, along with the IP address and port number of the predecessor and successor Jondoes are recorded in the Jondoe's routing table. If the coin flip indicates not to forward the message, the

Jondoe records a sentinel value for the successor hop in the routing table so it knows to decrypt the web request with the path key and forward it to R.

After the static re-routing path is created, web requests are sent by S's Jondoe by encrypting the web request with the path key, and the path ID with the symmetric key shared by the user's Jondoe and the successor's Jondoe (recorded in the routing table). Each intermediate Jondoe in the re-routing path decrypts the path ID with its shared key, looks up the successor Jondoe, encrypts the new path ID with the symmetric key shared by the intermediate Jondoe and the successor's Jondoe, and forwards the encrypted path ID and still encrypted URL request to the successor node. If a Jondoe looks up the successor for the path ID in the re-routing table and discovers the sentinel value for the successor hop, it decrypts the encrypted web request with the path key and makes the web request on behalf of the sender. It then sends the response of R on the backwards path similar to the method the web request was sent.

A.3 Buses Example: Buses

To further understand Buses, let us consider the following example. Since the optimal communication complexity protocol is rather simple and does not provide mutual anonymity, consider the optimal buffer complexity protocol. Suppose a sender S wants to send a message M to receiver R, the topology consist of five nodes S, N_2, N_3, R, N_5, and let the route of the bus form the ring S, N_2, N_3, R, N_5, S. In order for S to send the message to R, the path and transformations on the bus B consisting of seats s_1, s_2, s_3 are:

Assume S receives a bus B with all invalid seats:

1. S receives the bus and replaces all the seats with their respective decryptions: $B = D_{K_S^-}(s_1), D_{K_S^-}(s_2), D_{K_S^-}(s_3)$
2. S receives no valid messages.
3. S chooses a random seat to forward the layered encrypted message, without loss of generality, assume it is s_1. The construction of the seat is the message encrypted on the reverse path, i.e., encrypted iteratively with K_R^+, K_3^+, K_2^+ as follows:
   (a) m
   (b) $E_{K_R^+}(M)$
   (c) $E_{K_{N_3}^+}(E_{K_R^+}(M))$
   (d) $E_{K_{N_2}^+}(E_{K_{N_3}^+}(E_{K_R^+}(M))$
4. S forwards the bus $B = E_{K_{N_2}^+}(E_{K_{N_3}^+}(E_{K_R^+}(M)), D_{K_S^-}(s_2), D_{K_S^-}(s_3)$ to N_2.

N_2 receives the bus:

1. N_2 receives the bus and replaces all the seats with their respective decryptions: $B = E_{K_{N_3}^+}(E_{K_R^+}(M)), D_{K_{N_2}^-}(D_{K_S^-}(s_2)), D_{K_{N_2}^-}(D_{K_S^-}(s_3))$
2. N_2 receives no valid messages.
3. N_2 forwards the bus $B = E_{K_{N_3}^+}(E_{K_R^+}(M)), D_{K_{N_2}^-}(D_{K_S^-}(s_2)), D_{K_{N_2}^-}(D_{K_S^-}(s_3))$ to N_3.

N_3 receives the bus:

1. N_3 receives the bus and replaces all the seats with their respective decryptions:

   $B = E_{K_R^+}(M), D_{K_{N_3}^-}(D_{K_{N_2}^-}(D_{K_S^-}(s_2))), D_{K_{N_3}^-}(D^{-}_{K_{N_2}}(D_{K_S^-}(s_3)))$

2. N_3 receives no valid messages.

3. N_3 forwards the bus

   $B = E_{K_R^+}(M), D_{K_{N_3}^-}(D_{K_{N_2}^-}(D_{K_S^-}(s_2))), D_{K_{N_3}^-}(D_{K_{N_2}^-}(D_{K_S^-}(s_3)))$ to R.

R receives the bus:

1. R receives the bus and replaces all the seats with their respective decryptions:

   $B = M, D_{K_R^-}(D_{K_{N_3}^-}(D_{K_{N_2}^-}(D_{K_S^-}(s_2)))), D_{K_R^-}(D_{K_{N_3}^-}(D_{K_{N_2}^-}(D_{K_S^-}(s_3))))$

2. R receives a valid message in M and replaces the seat s_1 with random data.

3. R forwards the bus

   $B = s_1, D_{K_R^-}(D_{K_{N_3}^-}(D_{K_{N_2}^-}(D_{K_S^-}(s_2)))), D_{K_R^-}(D_{K_{N_3}^-}(D_{K_{N_2}^-}(D_{K_S^-}(s_3))))$ to N_5.

A.4 Broadcasting Example: DC-Net

To further understand the broadcasting technique consider the following example based on DC-Net. Suppose that the network G contain the nodes $V = \{N_1, N_2, N_3\}$, that the network is completely connected and contains the bi-directional communication channels $E = \{\{N_1, N_2\}, \{N_1, N_3\}, \{N_2, N_3\}\}$, and that each communication channel $\{N_i, N_j\}$ shares a secret key $K_{i,j}$: $K_{1,2} = (11001\ldots)_2, K_{1,3} = (01010\ldots)_2$, $K_{2,3} = (10101\ldots)_2$. Let node N_1 send the message $m = 1101_2$ to N_3 and immediately begins to send the message on the first observed parity.

The sequence of events is:

1. Round 1

   (a) N_1 calculates the parity of the first bit in its shared keys $1 \oplus 0 = 1$, Xors in the first bit of the message $1 \oplus 1 = 0$ and broadcasts 0.

   (b) N_2 calculates the parity of the first bit in its shared keys $1 \oplus 1 = 0$ and broadcasts 0.

   (c) N_3 calculates the parity of the first bit in its shared keys $0 \oplus 1 = 1$ and broadcasts 1.

   (d) All nodes calculate the parity of all broadcasted parities $0 \oplus 0 \oplus 1 = 1$ and every node that knows the first bit is being sent, know the first bit in the message is 1.

2. Round 2

   (a) N_1 calculates the parity of the second bit in its shared keys $1 \oplus 1 = 0$, Xors in the second bit of the message $0 \oplus 1 = 1$ and broadcasts 1.

   (b) N_2 calculates the parity of the second bit in its shared keys $1 \oplus 0 = 1$ and broadcasts 1.

   (c) N_3 calculates the parity of the second bit in its shared keys $1 \oplus 0 = 1$ and broadcasts 1.

   (d) All nodes calculate the parity of all broadcasted parities $1 \oplus 1 \oplus 1 = 1$ and every node that knows the second bit is being sent, knows the second bit in the message is 1.

3. Round 3

   (a) N_1 calculates the observed parity $0 \oplus 0 = 0$, Xors in the third bit of the message $0 \oplus 0 = 0$ and broadcasts 0.

   (b) N_2 calculates the parity of the third bit in its shared keys $0 \oplus 1 = 1$ and broadcasts 1.

   (c) N_3 calculates the parity of the third bit in its shared keys $0 \oplus 1 = 1$ and broadcasts 1.

   (d) All nodes calculate the parity of all broadcasted parities $0 \oplus 1 \oplus 1 = 0$ and every node that knows the third bit is being sent, knows the third bit in the message is 0.

4. Round 4

   (a) N_1 calculates the parity of the fourth bit in its shared keys $0 \oplus 1 = 1$, Xors in the fourth bit of the message $1 \oplus 1 = 0$ and broadcasts 0.

   (b) N_2 calculates the parity of the fourth bit in its shared keys $0 \oplus 0 = 0$ and broadcasts 0.

   (c) N_3 calculates the parity of the fourth bit in its shared keys $1 \oplus 0 = 1$ and broadcasts 1.

   (d) All nodes calculate the parity of all broadcasted parities $0 \oplus 0 \oplus 1 = 1$ and every node that knows the fourth bit is being sent, knows the fourth bit in the message is 1.

Appendix B

Bus Pseudocode

B.1 Bus Thread Pseudocode

The Bus thread is the parent of the Read, Write, and Acknowledgment threads. Thus, it is responsible for instantiating all the main shared thread objects and their respective mutual exclusion objects in addition to its own main objects. Let the pair *CS-START* … *CS-END* denote a critical section that must use mutual exclusion to prevent data races between the threads for shared objects.

```
1. Exit with error message if initialization parameter $k$ is larger than upper-bound $K$.
2. Instantiate global shared objects busQueue, forwardAckQueue, sendQueue, sentList, delTagList, and ackDataQueue and initialize them to be empty. Also, instantiate the global shared objects for mutual exclusion variables. In addition, instantiate the local reservedTagList and initialize it to be empty.
3. Spawn the Write thread, Read thread, and Acknowledgment thread.
4. while(true)
    (a) Wait for incoming bus.
    (b) CS-START Save the incoming bus to the busQueue CS-END.
    (c) If first bus received, insert $k$ random data seats and go to Step 4p.
```

(d) *CS-START* All entries in the *sentList* have the field, number of times nested message sent, incremented by 1. If this exceeds the threshold r for that message, or if the acknowledgment time-out has expired, a new indirection path is chosen and a new nested message is created (see Write thread Steps 1(b)i to 1(b)iv for pseudo-code) on the reversed indirection path. If the threshold r was exceeded, it is linearly increased. The seat containing the old nested message is deleted ensuring that not more than pk seats are deleted. In addition, maximum resend time-outs are checked, and if any are expired the node stops trying to deliver the nested message, deletes the corresponding seat ensuring that not more than pk seats are deleted, and delivers an error message to the application. Before encapsulating a new nested message (see step 4j for pseudo-code) in a seat to be resent, this step must ensure that it does not use more than $k - pk$ reserved seats (containing a valid message) in total; otherwise, the node will not be able to delete enough seats according to the p-threshold replacement back-off scheme. If the node is unable to resend the new nested message, it waits until the next pass of the bus and tries again. *CS-END*

(e) *CS-START* All entries in the *delTagList* have their delay field decremented. If any entries have $delay \leq 0$, then they are deleted along with their respective seat on the bus *CS-END*. The total number of seats deleted in this step and the previous step must not exceed pk for a single pass of the bus — any additional seats ready for deletion are deleted in a future pass of the bus.

(f) Any seats whose timestamps have expired are deleted. The total number of seats deleted in this step and the previous two steps must not exceed pk for a single pass of the bus.

(g) *CS-START* Update $reservedTagList$, removing any entries for seats that no longer appear on the bus *CS-END*.

(h) If fewer than pk seats have been deleted, seats that are not reserved should be randomly deleted until a total of pk seats, including those deleted in steps 4d, 4e, and 4f are deleted.

(i) *CS-START* As many encrypted nested messages as possible in the $forwardAckQueue$ are placed in seats on the bus *CS-END*. At most $k-pk$ seats can be reserved with valid messages in total; otherwise, the node may not be able to delete sufficient seats in the next iteration.

(j) To encapsulate the encrypted nested message into a seat and insert on the bus:

i. A new random 128-bit tag is generated.

ii. The seat constructor is invoked with the new tag, the node's public key, and the nested message. The seat is prepended with a seat header (public key, tag, timestamp) where the timestamp is generated upon construction. Also, random data is appended to the nested message to make it uniform in size. In addition, a signature is calculated on the 3-tuple {seat header, nested message, random data} and appended to the seat.

iii. The entire seat is appended to the bus, and the number of seats field

```
        for the bus is incremented by one.
    (k) CS-START As many encrypted nested messages as possible in the sendQueue are placed in seats on the bus as in Step 4j CS-END. At most $k - pk$ seats can be reserved with valid messages in total; otherwise, the node may not be able to delete sufficient seats in the next iteration.
    (l) If fewer than $k - pk$ nested messages have been placed on the bus in all the prior steps, random data seats are inserted so that the total number of modified seats for this iteration is $k - pk$.
    (m) Check that the total number of seats deleted is $pk$ and that there are no more than $k - pk$ reserved seats. Otherwise the $p$-threshold-replacement back-off scheme will fail on the next reception of the bus. If these conditions are violated, there is an error in the code and an error message is delivered to the application.
    (n) Ensure that the node still owns k-seats, inserting new random data seats as necessary.
    (o) If $busThreadWait == 1$, wait for Read thread to signal that all buses have been taken off the busQueue.
    (p) Choose the neighbour to forward the Bus to.
5. end while
```

B.2 Read Thread Pseudocode

```
1. Instantiate $receivedMessagesList$ and initialize it to be empty.
```

2. while(true)

   (a) *CS-START* Wait until *busQueue* is not empty.

   (b) Extract the next received Bus from the *busQueue*

   (c) If $busThreadWait == 1$ and the *busQueue* is now empty, signal the Bus thread. *CS-END*.

   (d) Decrypt all the seats on the Bus destined for the node.

      i. If *reduceDecrypt* is true, before decrypting each seat in the next step, check if the seat's hash is in the chained hash table. If the seat's hash is in the chained hash table, ignore the seat and do not decrypt or verify the seat (next 2 steps). If the seat's hash is not in the chained hash table, insert the hash of the seat into the chained hash table.

      ii. Apply hybrid decryption to each seat's encapsulated nested message with the node's private key and the anonymous private key and check for correct redundancy (valid).

      iii. Verify each valid seat's signature and deliver an error message to the application if a signature is incorrect.

   (e) After decrypting all the seats, if *reduceDecrypt* is true and the time-out *reduceDecryptInterval* has lapsed since the last time the chained hash table was clear, then clear the chained hash table.

   (f) Remove any elements in the *receivedMessagesList* that are older than *maxDuplicateTime*.

   (g) Check each of the valid and verified decrypted nested messages for replays.

```
        That is, if the forward field is true, then check if the 3-tuple {salt, timestamp, embedded nested message} is in $receivedMessagesList$, otherwise forward field is false so check if the 3-tuple {message tag, timestamp, message} is in $receivedMessagesList$.
    (h) For each of the decrypted nested messages that are verified, valid and not a replay:
        i. *CS-START* Add 2-tuple {tag,delay} to $delTagList$ where delay is randomly in $[0, D - 1]$ *CS-END*.
        ii. If the type of the nested message is an application message and forward is false, *CS-START* insert 2-tuple {message tag, anonymous PK} into *ackDataQueue* *CS-END*. Then, pass the message to the application.
        iii. If the type of the nested message is an acknowledgment message and forward is false, *CS-START* delete the entry from the *sentList* via the message tag *CS-END*.
        iv. If forward is true, then *CS-START* place the embedded nested message into the $forwardAckQueue$ *CS-END*.
3. end while
```

B.3 Write Thread Pseudocode

```
1. while(true)
    (a) Wait for a *message* and intended receiver $R$ from node's application.
```

(b) If R exists, *message* is valid, and message size will not result in a seat that exceeds the $uniformSeatSize$, create a nested message.

   i. Choose a dynamic indirection path of random length $l \in [1, L]$ consisting of the public keys $K_1^+, K_2^+, \ldots, K_l^+, K_R^+$.

   ii. Create $MessageHeader_R = (\text{salt}, \text{type}, \text{forward}, \text{message length}, \text{message tag}, \text{anonymous PK})$
   where salt is a random 128 bits, type is application (000_2), forward is false (0), message size is the size of the message being sent by the application (in bytes), message tag is a random 128-bit number, and anonymous PK is the anonymous public key of the sender.

   iii. Create
   $L_R = E_{K_R^+}(\text{salt}), E_{salt}(MessageHeader'_R, message)$ where $MessageHeader'_R$ is $MessageHeader_R$ without the salt.

   iv. For each K_i^+ for $i = (l-1), (l-2), \ldots, 1$ construct $MessageHeader_i$ and then L_i.
   $MessageHeader_i = (\text{salt}_i, \text{type}, \text{forward}, \text{message length}_i)$ where salt_i is a random 128 bits, type is application (000_2), forward is true (1), and message $\text{length}_i = sizeOf(L_{i+1})$ in bytes.
   $L_i = E_{K_i^+}(\text{salt}), E_{salt}(MessageHeader'_i, L_{i+1})$ where $MessageHeader'_i$ is $MessageHeader_i$ without the salt.

   v. *CS-START* Place nested message into *sendQueue* *CS-END*.

2. end while

B.4 Acknowledgment Thread Pseudocode

```
1. while(true)
   (a) *CS-START* Wait until *ackDataQueue* is not empty.
   (b) Extract entry from *ackDataQueue*. *CS-END*
   (c) Create nested message with type acknowledgment utilizing the message
       tag and anonymous PK from the *ackDataQueue* entry:
       i.   Choose a dynamic indirection path of random length $l \in [1, L]$ consisting
            of the public keys $K_1^+, K_2^+, \ldots, K_l^+, K_{anonymousPK}^+$.
       ii.  Create $MessageHeader_R = (\text{salt}, \text{type}, \text{forward}, \text{message length},$
            $\text{message tag}, \text{anonymous PK})$ where salt is a random 128 bits, type
            is acknowledgment (001), forward is false (0), message size is 0, message
            is NULL, message tag is from the *ackDataQueue* entry, and
            anonymous PK is the anonymous public key of the node.
       iii. Create
            $L_R = E_{K^+_{anonymousPK}}(\text{salt}), E_{salt}(MessageHeader'_R, message)$ where
            $MessageHeader'_R$ is $MessageHeader_R$ without the salt.
       iv.  For each $K_i^+$ for $i = (l-1), (l-2), \ldots, 1$ construct
            $MessageHeader_i$ and then $L_i$.
            $MessageHeader_i = (\text{salt}_i, \text{type}, \text{forward}, \text{message length}_i)$ where $\text{salt}_i$
            is a random 128 bits, type is acknowledgment (001), forward is true,
            and message $\text{length}_i = sizeOf(L_{i+1})$ in bytes.
            $L_i = E_{K_i^+}(\text{salt}), E_{salt}(MessageHeader'_i, L_{i+1})$ where
            $MessageHeader'_i$ is $MessageHeader_i$ without the salt.
         v. *CS-START* Place nested message into $forwardAckQueue$ *CS-END*.
2. end while
```