Implementation and Security Analysis of the Infranet Anti-Censorship System

by

Winston Wang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

Author . . . .
Department of Electrical Engineering and Computer Science
May 21, 2003

Certified by . . . .
Hari Balakrishnan
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Implementation and Security Analysis of the Infranet Anti-Censorship System

by

Winston Wang

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

As the Internet has grown in popularity, so have efforts to control and monitor its use. Countries such as China [46] and Saudi Arabia [25] already block certain kinds of content and more may follow as censorship tools become increasingly sophisticated. In response, Feamster et al. have proposed Infranet [10], a system allowing clients to retrieve sensitive content through a distributed network of secretly cooperating web servers. Infranet differs from other anti-censorship system in that it surreptitiously embeds requests and responses for sensitive content inside what appear to be innocuous web transactions. It is the first anti-censorship system that attempts to achieve covertness in addition to confidentiality.

This thesis presents several challenges that we encountered while implementing the first public release of Infranet. Our research has yielded a variety of innovative techniques to improve the speed, security, and scalability of Infranet. We describe a new technique for generating probabilitistic models of client behavior, as well as algorithms that leverage this knowledge to provide more efficient communication between clients and servers. We explore several attacks a censor might mount and propose defenses to counteract these measures. Our work is freely available [18] and can be built using open source tools for either Windows or Linux based systems.

Thesis Supervisor: Hari Balakrishnan
Title: Associate Professor

Acknowledgments

I would like to express my deepest gratitude to my advisor Professor Hari Balakrishnan, who has been a great source of inspiration and encouragement to me. I am continually amazed by the dedication and intelligence he possesses and am humbled by his professional success. His guidance has been invaluable and his support has allowed me to dedicate my time to this research for the past year.

I am also grateful to my colleague, Nick Feamster, who introduced me to the Infranet project and the Network Mobile Systems group. My time here would not have been nearly as productive without his generosity and insights. I thank him for all his help. I could not have asked for more interesting and challenging line of work.

I would like to thank my officemates, Bret Hull and Michael Walfish for making each day a little more enjoyable. I have been fortunate to be able to work in such an interesting environment.

Finally, I would like to thank my parents, as well as my brother Steven, for their support throughout the years. Everything I have accomplished I owe to them. They taught me the value of hard work and persistence. They were always there for me through the high and low points of my life. I dedicate this thesis to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The Internet has always offered the promise of a more connected world with the free exchange of information. It's pervasive and decentralized architecture can transcend government boundaries and regulations, making it difficult to attack and impossible to control. It is perceived by many to be a truly democratic forum where any voice could be heard, no matter how marginalized or controversial.

However appealing this notion may be, it does not reflect the state of affairs in the world today. In recent years, we have seen a growing conflict emerge between those who wish to impose control over the Internet and those who wish to maintain freedom, a battle which has increasingly favored the advocates of control. Faster hardware in routers and firewalls allow network administrators to shape and monitor their traffic in increasingly sophisticated ways. Companies regularly monitor their employees surfing habits and email. And the practice is likely to become more pervasive as technology becomes cheaper.

Governments have also leveraged these new technologies, attempting to block web traffic that is deemed to be offensive or revolutionary. A recent study by Harvard Law School researchers [46] found China to have the most extensive censorship, blocking up to a quarter of the most popular web sites on the Internet, ranging from Western media, human rights groups, Google, and even MIT. Saudi Arabia regularly block web sites deemed obscene [25] and the European Union has recently attempted to ban all hate speech on the Internet [28]. These issues are not isolated within national borders either. Recently, France appealed to the Supreme Court to remove offensive memorabilia from Yahoo auctions [28].

New developments signal evolving notions of privacy rights in the United States as well. Invoking the new Digital Millennium Copyright Act [9], companies have successfully forced ISPs to disclose private customer information in cases of suspected copyright infringement [27]. The Patriot Act

[33], passed through Congress in the wake of Sept. 11, allows government agencies broader power to monitor email and web browsing behavior without the normal procedures of due process. It remains to be seen how our government will choose to use these new found powers.

The examples above illustrate that censorship is an emerging and real issue that will have vast repercussion our society. In this thesis, we strive to further our understanding of the challenges this issue presents by presenting an implementation and analysis of a novel anti-censorship system. It is our belief that the free and open dissemination of ideas are vital to the preservation of a democratic society. However, this thesis will focus exclusively on the engineering and technological aspects of the issue, leaving the political, moral, and legal implications of this work for others to decide.

1.2 Overview

To counter this growing threat of censorship, Feamster et al. have proposed Infranet [10], a system that allows users to surreptitiously retrieve sensitive content through secretly modified HTTP servers globally distributed outside of the censor's network of control. Infranet is novel because it attempts to hide requests and responses for sensitive content within a sequence of seemingly innocuous web browsing transactions. Not only does Infranet protect the secret communications between Infranet clients and servers, but it additionally attempts to disguise its traffic in a way that masks its very existence. In short, Infranet presents the first such anti-censorship system which strives to achieve *covertness* in addition to confidentiality.

These unique features of Infranet provide valuable benefits to system users and significant obstacles to censors. Infranet gives its clients and servers a legitimate *plausible deniability* of using the system, reducing the risk of punishment and legal harassment for running Infranet. It hampers the ability of censors to methodically block or filter Infranet traffic since it is virtually indistinguishable from innocuous HTTP traffic. Likewise, discovering or tampering with Infranet sessions is significantly harder.

1.2.1 Design Goals

Infranet has been designed to meet a number of goals, not all of them complementary. Listed below are these goals, from the original Infranet paper [10].

1. *Deniability*: It should be computationally intractable to confirm that any individual is intentionally downloading information via Infranet, or to determine what that information might be.

2. *Statistical deniability*: Even if it is impossible to confirm that a client is using Infranet, an adversary might notice statistical anomalies in browsing patterns that suggest a client is using

Infranet. Ideally, an Infranet user's browsing patterns should be statistically indistinguishable from those of normal Web users.

3. *Server covertness*: Since an adversary will likely block all known Infranet server, it must be difficult to detect that a Web server is running Infranet simply by watching its behavior. Of course, any Infranet client using the server will know that the server is Infranet-enabled; however, this knowledge should only arise from possession of a secret that remains unavailable to the censor. If the censor chooses not to block access to the server but rather to watch clients connecting to it for suspicious activities, deniability should not be compromised. The server must assume that *all users are Infranet clients.* This ensures that Infranet client cannot be distinguished from innocent users based on the server's behavior.

4. *Communication robustness*: The Infranet channel should be robust in the presence of censorship activities designed to interfere with Infranet communication. Note that it is impossible to be infinitely robust, because a censor who blocks all Infranet access will successfully prevent Infranet communication. Thus we assume the censor permits some communication with non-censored sites.

   Any technique that prevents a site from being used as an Infranet server should make that site fundamentally unusable by non-Infranet clients. As an example of a scheme that is *not* robust, consider using SSL as our Infranet channel. While this provides full requester and responder deniability and covertness (since many Web servers run SSL for innocent reasons), it is quite plausible for a censor to block *all* SSL access to the Internet, since vast amounts of information remain accessible through non-encrypted connections. Thus, a censor can block SSL-Infranet without completely restricting Internet access.

   In a similar vein, if the censor has concluded that a particular site is an Infranet server, we should ensure that their only option for blocking Infranet access is to block *all* access to the suspected site. Hopefully, this will make the censor more reluctant to block sites, which will allow more Infranet servers to remain accessible.

5. *Performance*: We seek to maximize the performance of Infranet communication, subject to our other objectives.

1.2.2 System Architecture

The Infranet system architecture consists of requesters and responders which appear to take on the traditional roles of HTTP clients and servers respectively. In fact, nothing about Infranet should attract the suspicion of censors. Infranet responders act like ordinary web servers, returning content to Infranet and non-Infranet users alike. Transactions consist of valid HTTP requests and

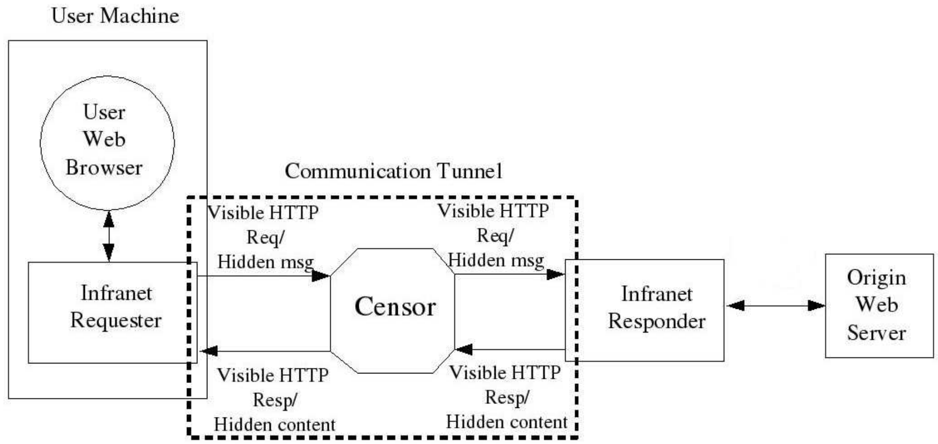

Figure 1-1: System architecture and threat model.

responses for legitimate URLs over ordinary TCP connections. Infranet does not modify any fields or headers of these protocols. Beyond the view of censors, though, Infranet requesters and responders communicate through a *covert communication channel* which is tunneled inside of ordinary HTTP transactions. In functionality, the responder acts like a web proxy. In the upstream direction, the requester transmits the user's covert request through the secret channel. The responder then fetches the request from the Web and, in the downstream direction, returns the sensitive content to the requester.

Requesters act as proxies between the user's browser and the outside Internet. When a client has started the Infranet software, the requester will intercept all requests from the browser. Instead of forwarding these requests to their (presumably blocked) servers, the requester will send the request to a corresponding Infranet responder through the secret channel. The subsequent response is passed back to the browser.

The design and implementation of this communication channel while preserving the illusion of innocuous web browsing is the most challenging technical aspect of Infranet. We give an overview of the basic concepts here with a more detailed description of the Infranet protocol in Chapter 3.

1.2.3 Upstream Communication

In the upstream direction, the requester communicates information to the responder by fetching a sequence of URLs in the visible domain which secretly corresponds to the client's hidden request in the covert domain. In other words, the URLs requested have additional semantics which are confidentially negotiated between the client and server. This prevents eavesdroppers from inferring the secret request.

It is also possible to embed upstream information into some of the unused headers of HTTP or

TCP. However, this solution is vulnerable to attacks since censors can easily remove or manipulate these fields without damaging innocuous clients. By using only the request URLs to convey information, censors can only disrupt upstream communication by dropping whole request packets or modifying URLs. Presumably, censors would be more hesitant to take such drastic actions to avoid disrupting innocuous traffic.

1.2.4 Downstream Communication

In the downstream direction, the responder uses *steganography* to hide data inside of images. Steganography is the technique of manipulating images in order to secretly embed information in them. These modifications distort the image too subtly to be detected by the human eyes. However, they are vulnerable to detection by more sophisticated statistical test, and can be destroyed by subtly cropping or distorting the image. Correcting these problems is an active field of research and beyond the scope of this thesis. However, we are confident that Infranet is amenable to a variety of data-hiding techniques.

1.2.5 Distribution

Before a requester can begin using Infranet, the requester must first be able to obtain the software. The traditional distribution technique, posting the software on a public web server, is problematic since downloading the software itself engenders suspicion. One solution to this problem is to distribute the software out-of-band, by CD or floppy disk. Another possible technique is to bootstrap Infranet onto an otherwise innocuous piece of software, preferably something popular and pervasive. Then the censor would not be able to block the software without blocking a popular service for its users. A third possible technique would be to embed chunks of the source code within popular music and video files and scatter them throughout the Internet via peer-to-peer file sharing networks. To prevent censors from placing their own malicious version of the code on the networks, an MD5 checksum of the source could be widely published, allowing users to distinguish the source from malicious imposters.

The requester must also be able to locate one Infranet responder. An overview of this *proxy discovery problem* is examined in Chapter 2, though it is not the focus of this thesis.

1.3 Contributions

This thesis documents the specific engineering and algorithmic challenges we inevitably encountered while developing the first public release of Infranet. We give an overview of the major contributions of this thesis.

1.3.1 Range Mapping

In the original Infranet paper, Feamster et al. [10] introduced a technique for upstream communication called *Range Mapping*, based on an asymmetric communication algorithm first developed by Adler and Maggs [1]. Range Mapping relies on the assumption that the server is aware of the *probability distribution function* of the client's secret request and exploits this knowledge to maximize the efficiency of the protocol.

In practice we have seen subtle problems in this approach that have forced us to radically overhaul these algorithms. The main issue we have encountered is that the dynamic nature of the Internet makes it impossible to accurately model the distribution of a user's request. We have introduced two novel techniques to solve this problem. First, we have developed an asymmetric communication algorithm designed to tolerate mistakes in the distribution, still guaranteeing convergence in the face of unexpected requests. We have proven the correctness of this algorithm.

Additionally, we have developed techniques for Infranet servers to learn from previous client behavior and generate a more accurate probability distribution, hence communicating more efficiently in the future. Infranet servers attempt to speculate future client requests by analyzing the contents of their previous requests. Servers also pay special attention to unexpected requests and take steps to handle similar requests more efficiently in the future. Collectively, these techniques are known as *Speculative Updating.*

1.3.2 Implementation

To prove the viability of our ideas, we have successfully developed a stable, robust version of Infranet that we have deployed on real web servers. Infranet software is built upon free, open-source API's and libraries and can be retrofitted to most web servers. We give a detailed description of the software architecture. We also present several key optimizations to our algorithms that significantly increase the performance of our system.

1.3.3 Security & Performance Analysis

We provide an extensive security analysis of the system, describing a variety of attacks that a censor might attempt and how Infranet can thwart these attacks. In many cases, we elaborate on the security analysis presented by Feamster et al. in the original paper, though geared specifically to the implementation we have created. We introduce a novel new technique that we have devised, entitled *Selective Degradation*, in which a censor can strategically disrupts network traffic in a manner that is more likely to harm Infranet clients than innocent users, even if the censor cannot identify the Infranet clients. We propose additional features to our protocol that allow Infranet sessions to recover from these malicious network errors, thereby mitigating the effectiveness of this attack.

Finally, we describe a series of experiments we have run on Infranet clients and servers to determine the performance of the system. These tests demonstrate that techniques presented in this thesis significant improve the effectiveness of our system.

Chapter 2

Related Work

From the Internet's inception, protecting the accessibility and freedom of its information have been active fields of research. Since wide-scale Internet censorship is a relatively recent phenomenon, early systems invariably tended to focus on user anonymity and privacy. However, these two problems are intractably linked to censorship and this early work serves as a foundation for many of the systems being developed today.

2.1 Anonymous Remailers

Much of the earliest research in enhancing Internet privacy was directed at *anonymous remailers* [14], special mail servers that allow people to send email without divulging their identity. Each remailer maintained a secret table mapping each user's real email address to a pseudonym. For each outgoing message, the remailer stripped out the real email address as well as any other identifying headers, and replaced it with the pseudonym. These early remailers were susceptible to a variety of attacks. The centralized location of the secret table made it vulnerable to social engineering attacks, subpoenas, and various forms of legal harassment. A passive attacker could infer a user's corresponding pseudonym by eavesdropping on remailer traffic and correlating incoming and outgoing messages.

A variety of technologies have been developed to combat these attacks. To obfuscate the source of a message, today's remailers often configure into a *mix-net*, an idea first developed by Chaum et al [5]. In this scheme, emails are encrypted several times with different public keys and are then sent through a series of remailers before arriving at their destination, each one decrypting the next hop with its private key. Since each intermediate remailer in the chain can only deduce the next and previous hops, the true source and destination of a message remain hidden to the intermediate nodes. This concept is often known as *onion routing* since each hop peals away one layer of encryption. To prevent correlation, messages are randomly reordered and padded with extra bytes. Remailers can also generate random cover traffic to further confuse eavesdroppers.

2.2 Web Proxies

Systems that protect web browsing anonymity operate in a comparable fashion. Organizations such as Anonymizer.com [2] and Voice of America [42] maintain well-known proxy servers that receive requests and forward them towards their destination, filtering out any user-specific information such as IP address and cookies in the process. As an additional measure, requests and responses are encrypted between the client and the proxy to prevent eavesdroppers from collecting private information.

Reiter et al. have developed another novel way of achieving anonymity in web browsing called Crowds [36]. Leveraging the notion of "blending into a crowd", users seek anonymity by banding together in groups. Requests are bounced around randomly within the group before being sent to the web server, disguising their true source. Crowds is also novel since it offers different levels of anonymity in exchange for performance.

It wasn't long before people realized that web proxies could be a simple circumvention tool for Internet censorship. However, web proxies suffer from one obvious weakness: the IP address of the proxy itself can be blocked as easily as any offensive web site. Triangle Boy [41] and Peekabooty [30] are two peer-to-peer systems in development that attempts to circumvent this weakness by using a set of distributed *reflectors* that serve as intermediaries, forwarding all requests to the well-known proxy. To protect the client from malicious hosts, Triangle Boy uses both SSL and certificates to ensure the security of requests and their trustworthy forwarding.

While this scheme make it more difficult for censors to block, it requires a mechanism for clients to learn the IP addresses of these intermediate forwarders. This problem illustrates one of the fundamental difficulties of anti-censorship systems, namely that any technique that a legitimate client can use to discover resources outside the censor's control can be used by the censor as well. The easier it is for clients to find outside hosts, the easier it is for censors to find them as well and block them. Triangle Boy's solution to this problem is to publish a constantly changing subset of the forwarders. In effect, the hope is to outrun the censors, publishing new forwarders faster than the old ones can be blocked.

Feamster et al. have addressed this issue from a slightly different perspective, which they have entitled the *proxy discovery problem* [11]. In this proposed scheme, once a client has learned the location of one proxy, it can then query the system to learn the location of others. Proxies are assigned in a manner that balances load across the system and prevents the censor from harvesting proxies in a methodical fashion.

Peekabooty has its own insights into this problem as well. They have suggested making the discovery a process that is easy for humans to do but difficult for computers, for example placing the names of forwarders inside images. Any human could view the image to discern the proxy, but technology does not yet exist for a computer to quickly parse out relevant text from an image. The censor would have to manually discover proxies. This would effectively make it a minor inconvenience to discover one forwarder, but a prohibitively tedious task to discover all.

2.3 Peer-to-Peer File Sharing

Freenet [6] is a peer-to-peer information storage system that allows users to anonymously post and retrieve data in a decentralized network. Content is stored and retrieved by globally unique keys, obtained by taking a secure, collision-resistant hash of the data. The advantage of using hashed keys is twofold: eavesdroppers cannot determine the nature of the requested data from the key, and the data tends to be distributed evenly among peers, making it fault-tolerant and resilient to attack. Like anonymous remailers, Freenet uses encryption and chaining in the propagation of searches, so that intermediate nodes cannot identify the source and destination of messages. This prevents censors engaging in two kinds of attacks: first, punishing those who download a piece of content and second, seeking out and dismantling all holders of a piece of content.

FreeHaven [8] is another proposed system for an anonymous, distributed file-sharing. Its unique design consists of publishers that trade *contracts*, guarantees for holding data for a length of time, along with a *reputation system* to ensure that servers meet their obligations. Tangler [43] is a different design, forcing publishers to replicate existing data on the system before it can publish its own data. This constraint, called an *entanglement*, forces all data to be replicated on the system, not just the most popular content.

2.4 Steganography

Steganography is the science of embedding secret messages within larger streams of data so that eavesdroppers cannot discern the presence or contents of the hidden messages. Like the field of cryptography, steganography concerns itself with allowing two parties to confidentially communicate over an untrusted network. However, steganography goes one step further, attempting to disguise the fact that any secret communication is taking place. In this respect steganography is closely tied with the goals of Infranet and it is no surprise that Infranet incorporates this technique.

The current generation of steganographic tools generally fall into two main categories. The first set of tools, considered to be the less sophisticated of the two, alter the least significant bits of the data. Tools like Hide and Seek [26], Mandelsteg [17], and White Noise Storm [3] utilize this technique. The second set of tools use either the wavelet or discrete cosine transformation (DCT) to locate areas of the image that can be safely modified without detection. Tools like Jpeg-Jsteg [15] and Outguess [32] follow this approach. Outguess improves on this basic framework in two ways. First, it employs a novel probabilistic technique to reduce the number of modified bits from the embedding process. Second, it identifies statistical anomalies caused by the embedding process and uses an error correcting transform to eliminate these anomalies [35].

The most common use of steganography is to track and verify the authenticity of data (also known as watermarking in this context). The subtle marks that governments place on their currency to protect it from counterfeiting can be considered a form of watermarking. The music industry has long considered digitally watermarking its audio files to trace music piracy. There have also been recent allegations that terrorist organizations like Al Qaeda use steganography to communicate their plans [24]. These claims remain unsubstantiated at this time [34].

Chapter 3

Infranet Protocol

Before discussing the new ideas we have researched, we must first give a detailed description the framework in which we are working, the Infranet protocol. As originally proposed by Feamster et al. [10], the Infranet creates a covert channel of communication tunneled through an ordinary sequence of HTTP requests and responses. During the main operation of the system, the protocol alternates between Upstream and Downstream Communication phases, exchanging secret requests and responses.

Additionally, each Infranet session is preceded by a Tunnel Setup phase. The purpose of this process is to negotiate a session ID and secret shared key between the Infranet client and server. The session ID exists so that the server can correlate incoming traffic with the correct client. The secret shared key is used to protect the confidentiality of the later Upstream and Downstream phases.

3.1 Tunnel Setup

HTTP is by nature a stateless protocol. In order to identify clients across multiple HTTP requests, the server must assign each client a session ID which is transmitted with each request. The most common technique for doing this is to store the session ID in a cookie. As an alternative, the server can dynamically rewrite all links in its HTML pages to embed the session ID into its URLs. The advantage of cookies is that they are easy to implement. The disadvantage is that they are easy to block. URL rewriting avoids this attack. However, it contains more computational overhead since it involves dynamically generating different web pages for individual clients.

The purpose of the Key Exchange phase of the protocol is to generate a *secret shared key* between the client and server to protect the security of all covert communication. However, this leads to a fundamental quandary: How can we securely communicate the very key needed to perform secure communication?

To solve this *bootstrapping problem*, each Infranet server generates its own public-private RSA

[37] key pair. The client is responsible for learning the public key out of band, presumably obtained through the same channel that the server's address was discovered. The client randomly generates the secret shared key and encrypts the data with the responder's public key. The secret key is then transmitted to the server using upstream communication with an initial key (again, transmitted out of band). The initial key provides an additional degree of protection against snoopers. However, since each client must be informed of the initial key to begin an Infranet session, it is more prone to fall into a censor's hands. However, even though the censor will be able to read what the client is transmitting to the server, without the private key the communication is incomprehensible.

3.2 Upstream Communication

Once the secret shared key has been transmitted, the Tunnel Setup phase is complete and the client and server can securely exchange information. During upstream communication, the client makes a sequence of visible HTTP requests, which correspond to a secret request string.

There are a variety of schemes for mapping between HTTP requests in the visible domain and the client's secret string in the covert domain. Each has its tradeoffs to performance, security, and covertness. We will discuss several such schemes.

3.2.1 Implicit Mapping

In the most simple case, we could convert the client's string to binary and assign half the URLs to map to 0 and half to map to 1. The client's stream of requests would then convey the binary representation of the client's secret string.

This is an example of an implicit mapping, where the client and server do not negotiate the semantics of visible URL, but instead the semantics are inferred somehow from the structure of the web site. This particular scheme provides good covertness. Since half the links can be permissibly requested from the client at any time, the client has wide latitude in its choice of requests, and hence can avoid suspicious browsing behavior. Unfortunately performance is less than optimal (n requests for n bits of data). Another glaring problem is security since the censor can just as easily infer the bits being transmitted. To increase the security of this scheme, the client could generate an RC4 stream cipher [38] from the secret shared key and encrypt the transmission with this cipher. Since only the server would have access to the secret key, eavesdroppers would not be able to infer the request.

We can also improve the performance of such a scheme by mapping URLs to more symbols. Instead of even and odd links, each link could correspond to a different fragment of bits, ideally transmitting $log(n)$ bits of information for each request where n represents the number of links. However, this performance gain comes at the cost of covertness since the client no longer has as

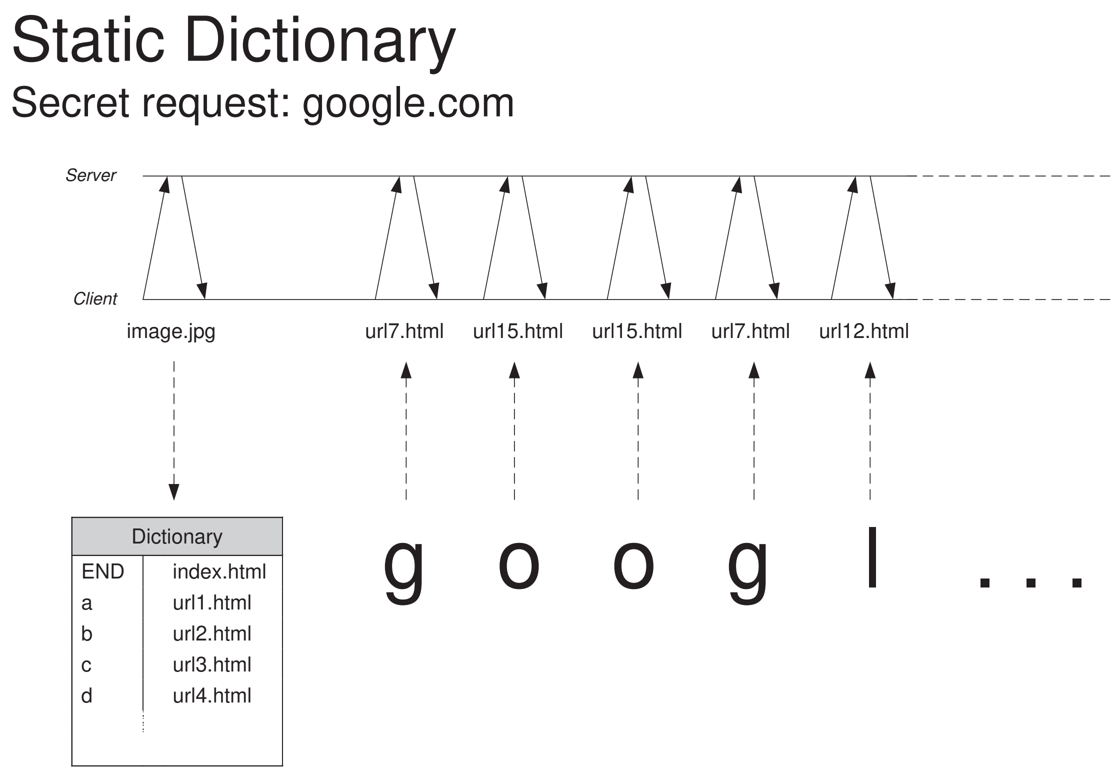

Figure 3-1: Example of static dictionary transmitting the secret string `google.com`

much flexibility in choosing its stream of requests.

3.2.2 Dictionary Schemes

Alternately, an Infranet responder can create a *dictionary* or *codebook* mapping fragments of strings to visible URL requests. This dictionary can be *static* or *dynamic* in nature. A static dictionary is transmitted one time at the beginning of the upstream communication and subsequently used to for all transmission. A dynamic dictionary is constantly updated over the course of the protocol.

The advantage of a static dictionary is easy implementation and faster performance since a new codebook does not need to be constantly generated and transmitted. However, a censor might notice statistical anomalies in browsing patterns, and worse yet, maybe be able to crack the codebook with enough requests and a database of common web sites. Therefore, the constantly changing dynamic dictionary has some advantages.

3.2.3 Range Mapping

In the upstream direction, the client only has its choice of URLs to convey information to the server, and hence can convey at most $log(n)$ bits where n represents the number of valid requests. In contrast, image downloads tend to be on the order of several kilobytes, with usually several hundred bytes of information available for manipulation through steganography. Therefore, the server can convey far more information in a single transaction than a client.

Systems that exhibit this *asymmetric communication model* were first studied by Adler and Maggs [1]. Infranet leverages their work in the creation of Range Mapping, an asymmetric communication algorithm specially tailored for Infranet's particular needs.

Range Mapping is essentially a guessing game between the client and server. In each iteration, the server sends a set S of tuples (s_i, γ_i). The client requests the URL γ_i that corresponds to the greatest such string s_i that is lexicographically less than or equal to the client's secret string. The server uses this information to provide a more refined list of split strings in the next iteration, progressively narrowing in on the client's secret string.

Range Mapping can provide significantly better performance than a dynamic dictionary scheme. The algorithm's main strength is that it exploits the probability distribution function of the client's secret string to make better choices for split strings and hence converge faster on more popular requests. Much of this performance analysis has been done by Adler and Maggs, who demonstrated that a client can transmit an n-bit string in significantly less than n iterations on average.

To minimize the number of iterations, the Infranet responder creates equal spaced thresholds throughout the probability distribution and chooses the split strings that conform to these intervals as closely as possible. This maximizes the amount of information conveyed in each iteration.

Equally significant is the fact that Range Mapping can be used to increase the covertness and statistical deniability of the system. Instead of creating equally spaced thresholds, the responder can manipulate these thresholds arbitrarily, in essence controlling the probability distribution of the client's requests in the visible domain. More formally, let us define a *link-traversal probability function* which represents "normal" behavior

$$p_{ij} = P(p_j \text{ is next request} \mid p_i \text{ is previous request})$$

The server can force the client to always conform to this model of behavior by choosing its threshold distances proportional to the appropriate link-traversal probabilities.

3.3 Downstream Communication

To perform downstream communication, Infranet leverages the technique of steganography, specifically the Outguess algorithm developed by Niel Provos [32]. Prior to communication with requesters, the responder computes offline the maximum amount of data that can be embedded into each image and store this information in a table.

When the responder has information that it wishes to transmit downstream, the client begins

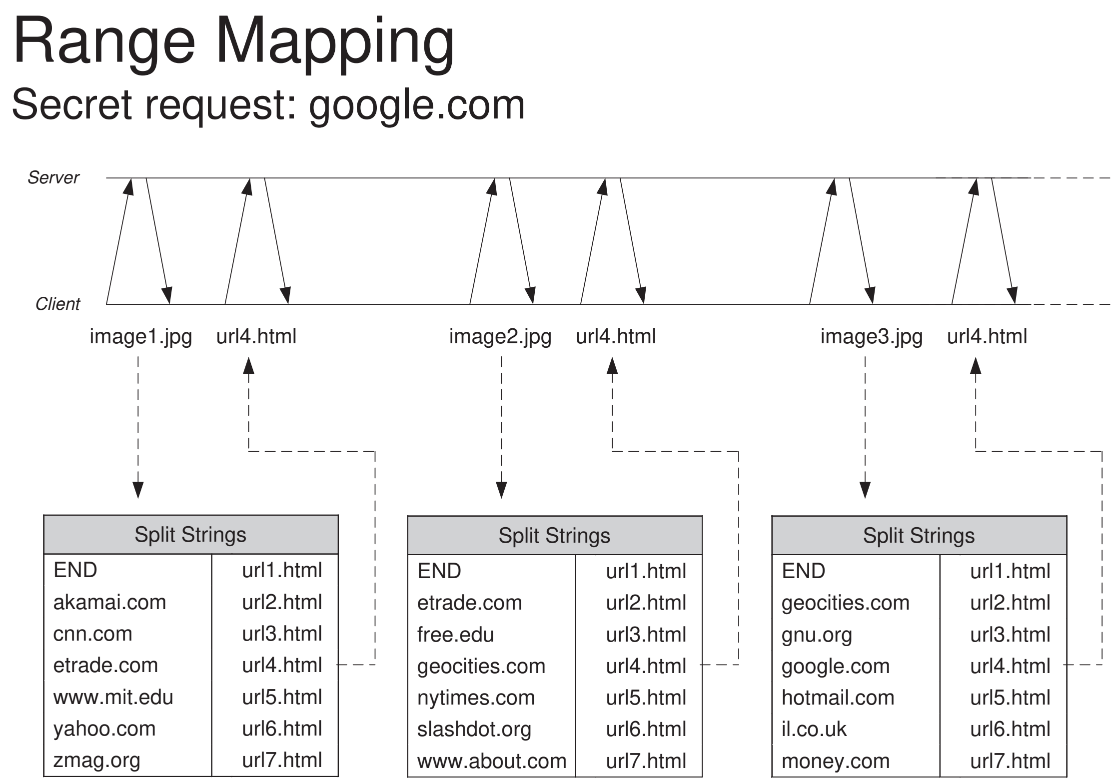

Figure 3-2: Example of the Range Mapping protocol transmitting the secret string `google.com`

downstream communication by requesting an image. The server checks the table to determine how much information it can embed into the response and breaks off a chunk of that size. Prepended to each chunk of data is a header indicating the size of the chunk, the size of the total message, and the relative offset of the chunk. Other information included in the header is the Z-bit, which is set when the server has compressed the message with gzip [16], and the version number. We envision future implementations may allow various upstream and downstream protocols for clients and servers with different needs.

Chapter 4

Range Mapping

The most technically challenging aspect of implementing Infranet is the design of the Range Mapping protocol, which is the focus of this chapter. Range Mapping is the technique used by Infranet clients to communicate secret messages to responders. As stated earlier, the protocol is a guessing game between client and server: the responder sends a list of possible strings, known as split strings, and the client indicates where its secret request lies with respect to these split strings. The server uses this information to provide a more refined list of strings in the next iteration. The protocol terminates when the server has zeroed in on the client's exact request.

Range-Mapping()

$str_l \leftarrow \varepsilon$
$str_h \leftarrow$ Max-String
while *true*
do $split \leftarrow$ Generate-Splits(str_l, str_h)
    Send-Split($split$)
    $i \leftarrow$ Receive-Client-Signal()
    if $i =$ Done-Signal
        then return str_l
        else $str_l \leftarrow split_i$
            $str_h \leftarrow split_{i+1}$

Range Mapping is itself modeled after a more general asymmetric communication protocol known as Bit-Efficient-Split, developed by Watkinson et al. [44] The principle behind Bit-Efficient-Split is that the server utilizes the probability distribution function of the client's message to carefully select which split strings to send. By the correct choice of split strings, the server maximizes the amount of information it obtains from the client in each iteration and converges quickly.

Bit-Efficient-Split()

$u \leftarrow 0, \quad v \leftarrow 1$

$y \leftarrow (min[A])^n, \quad z \leftarrow (max[A])^n$

while Split(u) $\neq$ Predecessor(Split(v))

do $\delta \leftarrow \frac{1}{N}(v - u)$

    for $i \leftarrow 1$ **to** $N - 1$

    do $split_i =$ Threshold-Split($u + \delta \times i$)

    Send-Split($split$)

    $j \leftarrow$ Receive-Client-Signal()

    $u \leftarrow u + \delta \times j, \quad v \leftarrow u + \delta$

return y

As point of reference, we present an high-level view of Watkinson's algorithm. In each iteration Bit-Efficient-Split creates a set of thresholds equally spaced throughout the interval and returns the associated *Threshold-Split* for each. Note that in other literature, what we refer to as the Threshold-Split is merely referred to as the split. We have renamed this concept to avoid confusion with other types of split strings that we will develop later.

Definition. *The k-Threshold-Split is the largest string s such that* $cdf(s) \leq k$.

Threshold-Split($thresh$)

$z \leftarrow \varepsilon$

while $\mid z \mid \leq n$

do $c \leftarrow max\{a \in A^* \mid \text{Cdf}(z \cdot a) \leq thresh\}$

    $z \leftarrow z \cdot c$

return z

For clarity, details from the paper have been omitted or modified. The definition of the split string is slightly different and in their presentation, only one split string is generated instead of N, for which the client makes a binary decision. The paper goes on to prove that a client can transmit an n-bit string with significantly less than n bits of communication. However, as we will see, the application of these ideas to the specific needs of Infranet present some very interesting questions.

We will begin by describing these questions and challenges in more detail. Following this, we present two preliminary solutions Uniform-Split and Common-Split which represent two disparate ways of approaching the problem. Next, we show how these two ideas can be reconciled to form a hybrid solution that captures both their strengths. We then introduce a technique called *Speculative Updating* that allows the system to dynamically personalize the system for each user, leading to more accurate guesses. Finally, we describe the syntax of the split string messages.

4.1 Challenges

Asymmetric communication protocols such as Range-Mapping and Bit-Efficient-Split can only be effective to the extent that the probability distribution function of the client request is accurately modeled. Therefore, it is vitally important to address key details such as what mathematical properties this distribution might have, how such a probability function could be computed, and what kind of data structures would be involved. In fact, this turns out to be a very open-ended problem with no clear answer. Three facts about URLs and the World Wide Web make representing an effective probability distribution a challenging problem.

- The World Wide Web is dynamic. Web sites appear and disappear daily. Existing web sites rise and fall in popularity, often times dramatically. There is no way to accurately document this changes in real-time. The World Wide Web is too chaotic.

- The web is gigantic, containing millions of web pages. Keeping track of every single possible URL that an Infranet client may request is unfeasible. Clearly, some precision must be sacrificed to fit practical computational and memory constraints.

- HTTP transactions are interrelated. URL requests are not independent. For example, a client who requests `www.google.com` heightens the probability of subsequently requesting `www.google.com/images/logo.gif`. Therefore, each user has a distribution function which is constantly changing over a normal browsing session.

The first point leads us to concede that it is not really possible to exactly model the distribution function of the World Wide Web. It is just too chaotic of a system. The second point leads us to concede that even if we could, it may not be practical to do so. The third point leads us to conclude that our distribution must not be static. The system must somehow tailor itself to different users and update itself dynamically in response to requests.

There are also aspects of Infranet that give Range Mapping a unique set of goals and requirements. While Bit-Efficient-Split strives to minimize upstream communication, in reality we have limitations in both upstream and downstream bandwidth which must be properly balanced.

4.2 Uniform-Split

Making no assumptions about the nature of URLs and web browsing behavior, one simple solution would be to model client requests as a uniform distribution over all possible finite-length strings. More formally, assume we have an alphabet A, composed of the permissible characters with some lexicographical ordering. Each character c has an associated probability $f(c)$ such that $\sum_{c\in A} f(c) = 1$ and a cumulative probability $F(c) = \sum_{a<c} f(a)$. The set of strings, A^*, represent all finite

sequences of characters. If we assume that the selection of each character of a string is independent then we get the following recursive definition of the cumulative distribution:

$$cdf(\varepsilon) = 0$$
$$cdf(c \cdot tail) = F(c) + f(c) \times cdf(tail)$$

Here ε represents the empty string and $x \cdot y$ represents the string concatenation operation. We will also use $|x|$ to denote the length of string x.

However, even such a simple model of client requests poses considerable difficulties. Bit-Efficient-Split assumes the transmission of a string whose length is known by both the client and server. This assumption additionally implies a finite universe of size $|A|^n$. Infranet must allow the client to sends request strings of arbitrary length.

With an infinite sized universe, the notion of the Threshold-Split is no longer well defined. Since there are an infinite number of strings less than a given threshold, no maximum string may exist. To illustrate this point, consider an alphabet $A = \{\mathtt{0}, \mathtt{1}\}$ with each symbol given a probability $\frac{1}{2}$. Consider the quantity:

$$k = \lim_{n \to +\infty} cdf((\mathtt{01})^n)$$
$$k = F(\mathtt{0}) + f(\mathtt{0}) \times (F(\mathtt{1}) + f(\mathtt{1}) \times k)$$
$$k = \frac{F(\mathtt{0}) + f(\mathtt{0}) \times F(\mathtt{1})}{1 - f(\mathtt{0}) \times f(\mathtt{1})}$$
$$k = \frac{1}{3}$$

Clearly, no such k-split string $s \in A^*$ exists.

Furthermore, it is no longer clear anymore how an algorithm like Bit-Efficient-Split will converge in an infinite universe. The general technique used is to maintain lower and upper boundaries u and v, such that $v - u \leq \frac{1}{N^i}$ after i iterations of the protocol. With a finite set of possible strings this will eventually reach the point where only one possible string is in the range of u and v. However, in our system there will always be an infinite number of strings within any interval.

Lastly, we must address the possibility of floating point roundoff errors in calculations. As Range-Mapping narrows in on the correct string, the split strings become closer and closer lexicographically and CDF calculations may no longer have an adequate amount of precision.

Presented here is a new algorithm, Uniform-Split that addresses these issues. But before delving into the specifics of Uniform-Split we first present a new concepts we have invented that will be central to the algorithm, the Interval-Split.

Definition. *Let $B = \{s \in A^* \mid u \leq cdf(s) < v\}$. The (u, v)-Interval-Split string is defined as the lexicographically largest string in B of minimal length.*

The Interval-Split represents a natural extension of the notion of the Threshold-Split. Whereas the Threshold-Split is defined as the largest string less than some threshold, this does not extend naturally to infinite domains. So all things being equal, we choose the shortest possible string that comes close to the threshold. Since we favor shorter strings to longer, we also get an added benefit to Infranet in reducing computational time and the size of downstream communications.

Uniform-Split(str_l, str_h)

$< prefix, t_l, t_h > \leftarrow$ Parse-Common-Prefix(str_l, str_h)
$\delta \leftarrow \frac{1}{N}(\text{Cdf}(t_h) - \text{Cdf}(t_l))$
for $i \leftarrow 1$ **to** N
do $u \leftarrow \text{Cdf}(t_l) + \delta \times (i-1)$
    $split_i \leftarrow$ Interval-Split($u, u+\delta$)
Prepend-To-All($split, prefix$)
return $split$

Interval-Split(u, v)

$str \leftarrow \varepsilon$
while $\text{Cdf}(str) \leq u$
do $c \leftarrow max\{a \in A \mid \text{Cdf}(str \cdot a) \leq v\}$
    $str \leftarrow str \cdot c$
return str

Our general strategy is to maintain upper and lower bound strings str_l and str_h. During each iteration of Range-Mapping, we take $cdf(str_l)$ and $cdf(str_h)$ and divide this interval up into N equal sized, contiguous intervals, and return the Interval-Split of each. We call this the uniform split of the range.

Property. *Let s_l and s_h be two strings such that $s_l < s_h$. Assume that both strings share a common prefix p such that $s_l = p \cdot t_l$ and $s_h = p \cdot t_h$. The uniform split of s_l and s_h is equivalent to the uniform split of t_l and t_h with p prepended to all resulting split strings.*

This observation, known as the Renormalization Property, directly stems from the fact that the selection of all characters in a string are independent in our probability model. This property is of practical importance to us since the longer split strings become over the course of the protocol, the higher the chance of floating point calculation errors mixing up the split strings. By cutting away a common prefix, this problem is avoided.

Theorem. *The technique of Range-Mapping using Uniform-Split to generate split strings will converge to the client's secret string.*

Proof. Let s be the client's secret string and $B = \{x \in A^* \mid x > s \text{ and } |x| \leq |s|\}$. B represents the set of all possible strings besides s that can be the split of an interval containing s. Since B is clearly finite, let t be the smallest element of B. Throughout the algorithm, we observe that $str_l \leq s < str_h$. If we can show that $t > str_h$ at some point of the algorithm, we can conclude that s is the Interval-Split of (str_l, str_h) and hence, the protocol has successfully located the secret string.

Let Δ_i represent the quantity $cdf(str_h) - cdf(str_l)$ during the i^{th} iteration of Uniform-Split by the Range-Mapping protocol. Since each split string resides inside contiguous intervals of size $\frac{1}{N}\Delta_i$, the maximum distance between any two split strings is $\frac{2}{N}\Delta_i$. Hence $\Delta_{i+1} \leq \frac{2}{N}\Delta_i$ which means that $\Delta_i \leq (\frac{2}{N})^i$, converging to 0 when $N \geq 3$. At some point of the protocol $\Delta_i < cdf(t) - cdf(s)$, implying that $t > str_h$. $\square$

Finally, there is one more minor point we need to address in order to make Uniform-Split fully functional. Returning to our previous example, consider $A = \{\texttt{0}, \texttt{1}\}$ with equal distribution. Since $cdf(\texttt{0}) = 0$ we observe that $cdf(\texttt{1}) = cdf(\texttt{10}) = cdf(\texttt{100})$... The fact that multiple strings may have the same cdf is problematic since Uniform-Split will never offer the client a string if there is a shorter one with the same cumulative probability. We can avoid this problem by adding a new symbol to our alphabet that is defined to be lexicographically less than all the other letters and given some small probability. This creates extra "space" between strings allowing our cumulative distribution function to distinguish between them.

4.3 Common-Split

Uniform-Split presents a model that does not fits what we know about the World Wide Web particularly well. Studies of Internet traffic show that the distribution of URLs is relatively skewed with a few popular web servers receiving a majority of the web traffic. Another approach would be to create a large table mapping between URLs and their corresponding cumulative distribution, ordered lexicographically. While we may not be able to place every possible URL within such a table, the fact that the Internet is so skewed towards certain common sites means that we could successfully service a large percentage of requests with a relatively small table. The downside is that the Range-Mapping algorithm would fail to locate the clients real request if it were not one of the common URLs in the table. We refer to this general approach as Common-Split.

More specifically, let C be the set of strings in the table and $common[s]$ represent the associated CDF for any $s \in C$. We define the following functions:

$$bound(s) = min\{x \in C \mid x \geq s\}$$
$$cdf(s) = common[bound(s)]$$

Common-Split does not present many of the challenges that we encountered with the development of Uniform-Split. Since the size of the table is finite, we avoid floating-point roundoff errors and problems with convergence altogether. However, Common-Split is not a direct application of the Bit-Efficient-Split algorithm.

Specifically, the creators of Bit-Efficient-Split assumed infinite bandwidth in the downstream direction and attempted to optimize upstream communication. However, Infranet must strike a balance between the two. Using the Threshold-Split is too "aggressive" for our purposes, converging very quickly to correct string, but generating very large split-string message for the client in the process. Ideally, we would like an algorithm that chooses split strings in a manner that may slightly decreases the amount of information conveyed in each iteration, but dramatically reduces the size of the downstream messages.

It turns out that we have already unknowingly developed such a concept, named the Interval-Split. Recall that the Interval-Split had a tendency to favor shorter strings in order to guarantee convergence on an arbitrary request. However, it also had the unintended positive consequence of decreasing computational time and downstream message size. We use Interval-Split in our Common-Split algorithm to retain these advantages.

There are a few additional minor problems that we must fix to ensure the correct operation Common-Split. We note that the Interval-Split is undefined in some situations since there may not be any strings in the distribution table between a given interval. To correct this, we define the Interval-Split to be the Threshold-Split in these situations. A second problem that we encounter is that the algorithm will sometimes produce a "bad" split string, which we define to be a string $split_i$ with no element $s \in C$ such that $split_i \leq s < split_{i+1}$. This problem does not seriously harm the operation of Range-Mapping, but it wastes the downstream bandwidth. We can easily identify and eliminate bad strings by testing whether $bound(split_i) = split_{i+1}$.

Through experience testing the system we have also made the observation that Interval-Split can occasionally be a little bit too "conservative" about its split strings selection. Therefore we add a second exception. If there is only one string s between $split_i$ and $split_{i+1}$, we set $split_i = s$. This saves us one iteration of the protocol in a few cases at very little cost.

It may seem like the differences between Uniform-Split and Common-Split are minor at best. Relating to many of the theoretical concepts, this is true. We did not describe our algorithm or prove its correctness since it is essentially the same as Uniform-Split in behavior. In practice, though, the implementation of Common-Split presents a variety of additional engineering challenges. Since the common table tends to be large (≈200,000 entries), the cumulative strain of thousands of lookups tends to slow performance of the system noticeably. Once we add in some of the clever tweaks that we have introduced to the algorithm, Common-Split and Uniform-Split are be barely recognizable.

Performance takes on a dual importance with respect to Infranet. An inefficient algorithm, in addition to creating a frustrating experience to users, poses a security threat as well. A server that periodically takes an unusual amount of time to service a simple request could be perceived as suspicious to a diligent censor. The main computational bottleneck in Infranet servers is the large number of table lookups that are involved in generating a list of split strings. Therefore we seek to remove duplicate or unnecessary query.

The first observation we make is that the algorithm generates multiple split strings for each iteration of the algorithm. Oftentimes, the calculation of different split strings will contain overlapping queries to the common table. The first optimization we have implemented, referred to as *amortization*, is a technique to reuse this redundant information, thereby lowering the amortized cost of split string calculations.

Another observation we make is that Common-Split uses a distribution that increases at discrete points in a step-wise fashion. If the common table queries are clustered tightly, many lookups could fall between the same two table entries and thus return the same value. We know that for all strings s and t that if $s \leq t \leq bound(s)$ then $cdf(t) = cdf(s)$. Therefore, the algorithm can save itself the lookup of t if it has already looked up s in the past. The implementation of this technique is referred to as *boundary checking*.

We will defer further discussion of *amortization* and *boundary checking* until section 5.1.4 in the Implementation section. We believe that they need to be described at a lower level of abstraction to fully appreciate.

4.4 Hybrid-Split

Uniform-Split and Common-Split have complementary attributes. Using Uniform-Split is a slow and methodical technique that is guaranteed to discover any string. Common-Split locates certain strings very quickly but cannot guarantee success.

To reconcile Uniform-Split and Common-Split, the Range-Mapping protocol uses a combination of the two techniques, a technique we call Hybrid-Split. At the beginning of the Range-Mapping protocol, the server uses Common-Split to determine which strings to send to the client. If the client's secret string is in the common table, the protocol will successfully identify the string and terminate. However, if the protocol reaches a point where no table entry is in between the lower and upper bound strings, the protocol will switch over to Uniform-Split and continue from there.

4.5 Speculative Updating

Up until now, we have assumed that each client's request is independently drawn from a static probability distribution. This is incorrect. There are definite patterns in normal browsing behavior that the system can exploit, allowing it to guess with a high degree of accuracy what a user might download next. This technique, entitled *Speculative Updating*, improves the efficiency of the Range Mapping protocol. This section discusses the two main challenges of implementing this technique, namely, accurately predicting future client behavior, and incorporating this data into the probability distribution. We address these issues in turn.

How can Infranet servers guess future requests? One observation we make is that in a typical browsing session, a majority of the URLs are not directly generated from the user: they come in the form of links, images, redirects, and a host of other tags specified in HTML files. In order to learn these URLs, the Infranet server can quickly parse through the client's HTML files before sending them back downstream, searching for any attributes that could be a potential future requests. Additionally, the server will sometimes need to perform some mundane manipulations on these strings, like prepending the server and path if the URL is relative.

One class of requests, input driven forms, are trickier to handle since these requests contain a *query* field with user generated data. This portion of the URL is nearly impossible to predict. More specifically, we face a situation where the Infranet server knows the prefix of a possible future request but cannot match the entire string. The ideal behavior in this case would be for the Range Mapping protocol to converge on the known prefix very quickly and then switch over to the Uniform-Split algorithm immediately to receive the unknown section of the URL. It turns out that we can coax this behavior out of the protocol without modifying the algorithms or data structures of our system. The trick is to add the desired prefix p to the distribution along with its associated *prefix delimiter*, as defined below.

Definition. *The prefix delimiter of p is defined to be the smallest string q such that $q > p \cdot x$ for all strings x.*

Note that the prefix delimiter is not always guaranteed to exist, in which case we only add p to our distribution. We also show the algorithm for calculating the prefix delimiter. The Successor function is defined to return ε for any character with no successor (i.e. the highest character).

Prefix-Delimiter(s)
$t \leftarrow \varepsilon$
for $i \leftarrow n$ **downto** 1
do if $|t| = 0$
    then $t \leftarrow$ Successor$(s[i]) \cdot t$
    else $t \leftarrow s[i] \cdot t$
return t

Parsing HTML files is a very effective strategy for predicting the client's future requests. However, the system will still see unexpected requests on a regular basis. Mainly, these requests are generated by a scripting language that the Infranet server cannot interpret. Occasionally, web sites use some of the more obscure features of HTML, or the client has typed a new site into its browser.

Can we learn any information from these requests? One general observation that we have made is that unexpected requests tend to come in bursts. We also notice that the URLs of these requests tend to be similar, the likely rationale being that whatever mechanism generated the unexpected request in the first place is also generating the subsequent requests. Since unexpected requests are relatively expensive to the system, it makes sense for Infranet servers to pay extra attention to these strings.

Unexpected requests mostly come from embeds web scripts and hence tend to be dynamically generated. Generally, the syntax of these requests is a static prefix ending with a separating character, followed by the dynamic portion of the URL. Common separating characters are the forward slash, the question mark, the ampersand, and the semicolon. The Infranet server does not attempt to determine which part of the URL is static and which part is dynamic. It merely looks for all prefixes that end in a common separator and adds these prefixes to the table. It is a simple strategy, and perhaps a little bit too aggressive, but the cost of adding too many strings to the distribution table is generally offset by the number of iterations saved when a prefix hits.

To effectively implement Speculative Updating we need to a strategy to efficiently manage dynamic table updates. The task of adding entries to the common table in Common-Split clearly becomes problematic as the table gets larger. It involves moving large blocks of memory around and updating the CDF of many table entries. Additionally, since each user is adding different strings, a different copy of the table needs to be kept for each user, severely taxing system resources. Therefore, Infranet implements a dual table approach. Common-Split retains one table which is large and immutable, containing the most common URLs that are likely to be requested by all users. We supplement this with a user table, one of which is associated with each client. Each table contains the list of URLs that its corresponding user is likely to request next.

The user table contrasts with the common table in that it is expected to quickly handle dynamic updates. To keep the computational cost of updates low, the user table manages its size by setting a hard cap on the number of entries it may have. To enforce this cap, the user table contains a timestamp for each entry. When the table exceeds its maximum size, the system discards the oldest strings and recomputes the cumulative distribution of the remaining table entries.

The common and user tables are never physically combined into a single, unified distribution. Instead, when calculating the cumulative distribution of strings, the system queries both tables and takes a weighted, normalized sum of the two responses.

```
BIND /mail-archives/cm/0041.html 46
BIND /mail-archives/cm/0072.html 77
BIND /mail-archives/cm/0031.html 36
49 END
77 POST
ROOT www.y
46 a
92 ahoo.com/
6 ahoo.com/r/i
72 ahoo.com/r/m1
90 ahoo.com/r/m2
73 ahoo.com/r/s
48 ahoo.com/s
31 ahoo.f
36 el
18 ep
58 i
```

Figure 4-1: Example split string file.

4.6 Message Format

The Infranet responder generates a list of split strings and their corresponding signal URLs and embeds them into images for downstream communication. The data format of the split strings has been designed to efficiently use space.

Figure 4-1 shows a set of split strings representing possible guesses for the user's hidden request and their associated signal URLs. We will call these the signal URLs (to distinguish from the split strings, which represent the request URL). The syntax for split string files is as follows.

BIND *alias signal* Signal URLs are often repeated over the course of an Infranet session. Therefore, it makes sense to have a shorter string that serves as an alias for the signaling URL to minimize message size. The alias is assumed to be valid for the life of the session and does not need to be redefined.

alias split Tells the client to request the signal URL associated with the *alias* if *split* is the lexicographically largest string less than or equal to the secret string.

ROOT *prefix* Split-strings tend to become closer together lexicographically, especially after the several iterations of the Range-Mapping algorithm, and often contain the same prefix. This command indicates to the client to prepend *prefix* to all splits-strings, allowing split strings to be stored in less space.

***alias* POST** Infranet allows the user to make two different kinds of requests: POST and GET. A GET action consists of sending a request URL. A POST action consists of sending a request URL with some additional data, usually populated from input forms on a web page. The user requests the alias with POST to signal that the request URL has been successfully found and to begin transmission of the POST data.

alias **END** The client requests the signal URL to terminate the Transmit Request phase and to proceed with the Transmit Response phase of the protocol.

Chapter 5

Implementation

We first describe the technologies used to implement our version of Infranet, followed by a description of the software architecture, APIs, and a discussion of the challenging and novel implementation issues encountered.

5.1 Server

The server is implemented in Java [22] and deployed on Apache's Tomcat [40] web server. First, we briefly describe Java API for writing web applications (known as J2EE [21]).

Architecturally, creating server-side web applications consists of developing Servlets, i.e. objects which are passed request objects and write the subsequent output data to response objects. For those knowledgeable about the Apache web server, Servlets are roughly equivalent in functionality to modules. In version 2.3 of the Servlet specification, Sun introduced the concept of Filters. Filters serve as wrappers around Servlets that can perform modifications to the request or response before or after the Servlet executes.

The Filter is an ideal implementation for Infranet since our system strives to appear and act like a regular web server, merely adding bits to certain responses and performing extra book-keeping tasks for the additional semantics associated with requests. The Filter model allows us to wrap Infranet around the original behavior of the web server without modification.

The Servlet-Filter model completely eliminates a large class of possible security weaknesses in the development of Infranet. For example, the alternative to a Filter would be to write Infranet as a Servlet which reproduces the behavior of the original web application. However, it is important to note that the default behavior varies among different web servers. They all react a little bit differently to erroneous or malformed requests and subtly interpret request headers differently. Therefore, different implementations would be need written for Tomcat, Websphere [45], iPlanet [19], and every other web server with a Java Servlet engine. Apache, the most common web server does

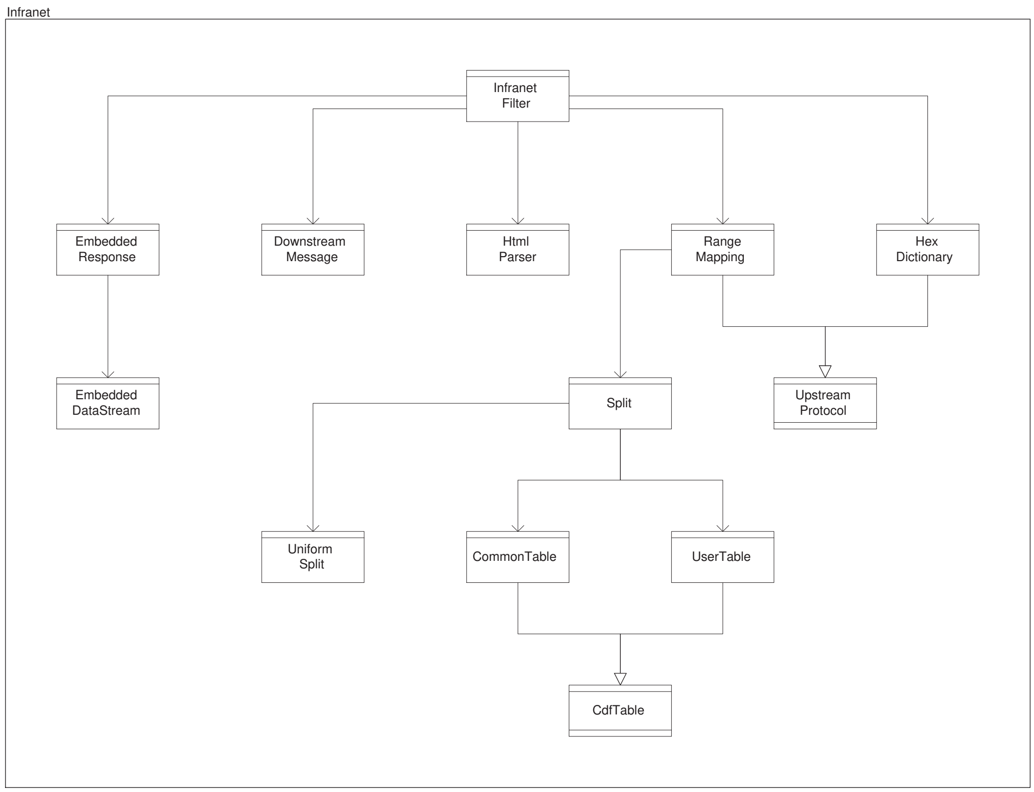

Figure 5-1: Module dependency diagram for the server software architecture

not by default with a Java Servlet engine, but many free plug-ins exist. With Filters, the original behavior is always invoked and one code base can be maintained.

The J2EE API also provides a host of other conveniences. The API contains a generalized Session object which serves to maintain state across multiple HTTP transactions from a single client. This API allows the system to easily remember client specific objects and variables between HTTP requests without having to worry about the underlying details of managing cookies, synchronization, or interprocess communication between multiple forked instances of the web application.

The Java Runtime Environment eases cross-platform deployment. Tomcat is free, open-source, and lightweight. This is important since most web server operators will be running Infranet for altruistic reasons. They will only deploy the system if it is free and easy to configure and maintain.

5.1.1 The Infranet Filter

As described earlier, the Infranet architecture is designed in such a way that Infranet code can be executed without disturbing the underlying web application. Typically, when an application server receives a request, it will invoke the appropriate Servlet, passing along a request and response object. The Servlet will then read the request object and write the appropriate data to the response object, which will typically feeds the data directly back to the client. The InfranetFilter class performs two

```
public class InfranetFilter implements Filter {
  public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) {
    HttpSession session = request.getSession();
    String uri = req.getRequestURI();

    ClientState clientState = (ClientState)session.getAttribute("clientState");
    if (clientState == null) {
      clientState = new ClientState();
      session.setAttribution("clientState", clientState);
    }

    boolean filterCalled = false;
    if (!clientState.message.isEmpty()) {
      Integer size = (Integer)maxBytes.get(uri);
      if (size != null) {
        byte[] chunk = clientState.message.getChunk(size);
        EmbeddedResponse embed = new EmbeddedResponse(res, clientState.key, chunk);

        filterCalled = true;
        chain.doFilter(request, embed);
        if (!embed.success()) {
          clientState.message.undo();
        }
      }
    }
    if (!filterCalled) chain.doFilter(req, res);
    updateState(clientState, uri)
  }
}
```

Figure 5-2: Simplified view of the Infranet Filter

main functions on top of this. First, it intercepts all the requests and updates any Infranet specific variables. Second, it hands to the Servlet a modified version of the response object that will receive data from the Servlet and steganographically embed secret messages into the stream before sending it back to the client.

Figure 5-2 shows the InfranetFilter source code, simplified for readability. The filter first loads up the client's state. Notice that the filter does not need to parse or set any cookies in order to associate the correct clientStates with the correct requesters. At runtime, the web application notes that the filter uses session specific data and automatically handles the necessary steps.

Second, the filter checks if there are any pending downstream messages and if so, how much information can be embedded into the client's request. The function `message.getChunk()` meters out an appropriately sized chunk of bytes with the correct downstream header prepended. `EmbeddedResponse` represents the modified response object that is passed along to the Servlet. The Servlet is invoked with `chain.doFilter()` and the session variables are updated in the method `updateState()`.

An Infranet session can be represented by a simple finite state machine. Figure 5-3 shows the four variables that make up the `ClientState` class. The state field represents where the client is in the Infranet session. It takes one of four possible enumerated values: `SHARED_KEY`, `UPDATE`, `DEMOD`, and

```
private class ClientState {
  public int state;
  public String key;
  public UpstreamProtocol prot;
  public InfranetMessage message;

  public ClientState() {
    state = SHARED_KEY;
    key = initialKey;
    prot = new HexDictionary();
    message = prot.getDownstream();
  }
}
```

Figure 5-3: Representation of the client's state

`SERVE`. The key field represents the key used for communication. In the beginning the key is set to whatever initial value has been specified by the Infranet server's owner. After the `SHARED_KEY` phase has completed, it is set to the secret shared key. `UpstreamProtocol` is a class representing the style of upstream communication that the Infranet protocol is currently using. The pending downstream message is stored by the `message` field.

Figure 5-4 shows the operation of the finite state machine. During the creation of the client's state, the state is initialized to `SHARED_KEY` and the upstream protocol is set to a static dictionary especially designed for conveying a hexadecimal string. After the server has finished receiving the secret shared key, the responder next engages in upstream communication with the client, alternating between the `UPDATE` and `DEMOD` phases. During the `UPDATE` phase, the server transmits the set of split strings to the client. In the `DEMOD` phase, the client requests a URL selecting the appropriate range. Finally, when the secret request has been determined, the protocol shifts to the `SERVE` phase, where the secret response is returned to the client.

5.1.2 Upstream Protocol

To provide a layer of abstraction between the Infranet filter and the choice of upstream communication, the `InfranetFilter` does not deal directly with the details of the upstream protocol. It maintains a class of type `UpstreamProtocol` on which it merely calls the function `setUpstream(...)` to pass the requested URL onwards, letting the `UpstreamProtocol` decide how to interpret the URL. If Infranet is using a protocol that requires sending information to the client at any point, the filter calls `getDownstream()` to fetch this data. Finally, the server determines when the upstream protocol terminates by calling the `isDone()` function.

`UpstreamProtocol` is an interface that is implemented by two classes, `HexDictionary` and `RangeMapping`. `HexDictionary` is a static dictionary with an alphabet of only 16 characters. It is solely used in the `SHARED_KEY` phase of Infranet to receive the secret shared key. Subsequently, all upstream communi-

```
private void updateState(ClientState client, String uri) {
  switch(client.state) {
  case SHARED_KEY :
    client.prot.setUpstream(uri);
    if (client.prot.isDone()) {
      client.key = getStegoKey((String)client.prot.getMessage());
      client.prot = new RangeMapping();
      client.message = new InfranetMessage(client.prot.getDownstream());
      client.state = UPDATE;
    }
    break;
  case UPDATE :
    if (client.message.isEmpty()) {
      client.state = DEMOD;
    }
    break;
  case DEMOD :
    boolean success = client.prot.setUpstream(uri);
    if (client.prot.isDone()) {
      Request r = (Request)client.prot.getMessage();
      byte[] data = fetchRequest(r);
      client.prot.addGuesses(HtmlParser.parseHtml(r, data));
      client.message = new InfranetMessage(data);
      client.state = SERVE;
    }
    else {
      if (success) {
        client.message = new InfranetMessage(client.prot.getDownstream());
        client.state = UPDATE;
      }
    }
    break;
  case SERVE :
    if (client.message.isEmpty()) {
      client.state = UPDATE;
      client.prot.clear();
      client.message = new InfranetMessage(client.prot.getDownstream());
    }
  }
}
```

Figure 5-4: Source code for Infranet's method of updating client state cation uses `RangeMapping`.

`RangeMapping` performs all of the book keeping associated with the protocol. It does not calculate the split string algorithms directly, instead relying on the `Split` class. Mostly, `RangeMapping's` main task is to take the raw split strings generated by the split class, associate each one with the correct URL in the visible domain, and format the data in the correct syntax for the client.

5.1.3 Common & User Tables

To represent the probability distribution of client requests, Infranet maintains classes `CommonTable` and `UserTable`, analogous to the common and user tables introduced in Sections 4.3 and 4.5.

`CommonTable` contains a singleton instance of a `TreeMap`, which associates `Strings` (possible client requests) to `Doubles` (cumulative distributions). The `TreeMap` is a data structure built into the Java API that allows fast lookup on a map sorted by key. It is implemented using the Red-Black tree algorithm.

`UserTable` is responsible for storing the URLs that the system thinks that a client may request next. Unlike `CommonTable`, `UserTables` are not shared between all users and hence, steps must be taken to limit their size. As URLs are added to `UserTable`, they are stored in a FIFO queue represented as a linked list. After the size of the linked list crosses a prescribed threshold, the `UserTable` begins discarding the oldest entries. Before the split string algorithm is invoked, `UserTable` generates a TreeMap in the same manner as `CommonTable` for easy lookup. The weight assigned to each URL is constant.

Both `UserTable` and `CommonTable` contain two main functions used by the split string algorithm. The function `public double cdf(String s)` returns the cumulative distribution of `s`. The function `public String bound()` returns the bound of the most recent cdf query.

5.1.4 Split Algorithm

In Chapter 4, we extensively discussed the Common-Split and Uniform-Split algorithms that form the basis of Range Mapping. However, the task of implementing an optimized version of Common-Split in Java presents enough interesting tweaks and tricks that at first glance it is not entirely clear that there is a relationship between the theoretical algorithm described in Section 4.3 and the Java implementation shown in Figure 5-5. This section seeks to bridge this gap. We will omit discussion of the Uniform-Split algorithm since the implementation is fairly straightforward.

For convenience, let $split_i$ = *Interval-Split(*`thresh[i-1]`*,* `thresh[i]`*)*. The algorithm calculates each split string $split_i$ by maintaining a running buffer `out[i]`. Each execution of the search block locates the first character s such that `cdf(out[i] + s) > thresh[i]` and then appends the preceding symbol, `out[i].append(r)`. The algorithm then checks whether the buffer meets the requirements of a valid split string (as calculated by the boolean variable `notDone`), and if so, moves onto the next split string. Otherwise, it continues searching for another symbol to append to `out[i]`. Note that each symbol is not represented as a `char`, but instead as a `String` of length 1. To get the actual character associated with a symbol `s`, we call `s.charAt(0)`.

The first optimization we present is the `Bound` class, which in functionality helps the split algorithm perform queries on the user and common tables. The following commands all produce the same value:

```
1.) x = COMMON_RATE * commonTable.cdf(s) + USER_RATE * userTable.cdf(s);
2.) x = new Bound(s).cdf;
3.) Bound b = new Bound(anyString);
    x = bound.update(s).cdf;
```

However, using `Bound` is better than a direct table query since `Bound` and `Split` have been carefully designed to work together to avoid unnecessary lookups through the *boundary checking* technique first described in Section 4.3. The `update()` function does not automatically query the common and user tables. It first checks the boundaries of the previous lookup it performed and avoids a query if possible. The `bound` variable provides an additional service to the algorithm. As the algorithm searches for more symbols to append to `out[i]`, the field `bound.bound` tracks the greatest known string s such that $s \leq split_i$.

Loop Invariant. `out[i]` $\leq$ `bound.bound` $\leq$ `thresh[i]`.

How does this optimize the algorithm? Recall that the purpose of the search block is to locate the first symbol s such that `cdf(out[i] + s) > thresh[i]`. If `out[i] + s` $\leq$ `bound.bound`, we can immediately discard `s` as a possibility, saving one table lookup. In fact, we can make the algorithm a little bit faster by avoiding this string comparison as well. Since `out[i]` is always a prefix of `bound.bound`, it suffices to compare `s` to the character of `bound.bound` at index `out[i].length()`, represented in our algorithm by the variable `borderChar` (in Java, array indexing starts at 0).

We can further optimize our algorithm from the fact that we are not restricted to calculating one split string at a time. If the algorithm can infer information about $split_{i+1}$ in the course of calculating $split_i$, the algorithm can exploit this knowledge to reduce the total computational time, a technique we refer to as *amortization*.

The implementation of this optimization is deceptively simple. Once the algorithm has located the correct symbols `s` and `r`, it appends `r` to `out[i]` but additionally checks whether the CDF of `out[i] + s` crosses any subsequent thresholds. If so, it appends `r` to those buffers as well. There is also a border case that the algorithm deals with, in which no symbol crosses `thresh[i]`. In this case, the algorithm appends `r` (the highest character) to all subsequent buffers of equal length to `out[i]`. It may not be entirely clear why this leads to correct behavior. To understand how the algorithm works, we prove the following loop invariant.

Loop Invariant. *For all $j \geq$ `i`, `out[j]` equals the longest matching prefix between `out[i]` and $split_j$.*

Proof. First, we state some obvious properties of the longest matching prefix function. Given any strings p and q and symbol c, the following statements hold:

1. $lmp(p, q) = p$ if and only if p is a prefix of q
2. if $p \cdot c$ is not a prefix of q then $lmp(p \cdot c, q) = lmp(p, q)$

At the beginning of the algorithm each buffer is initialized to the variable `given`, representing the longest matching prefix between `lower` and `upper`. Since every string between `lower` and `upper` begins with `given` our invariant holds.

We note that the buffers are only manipulated in the block labeled `search` and then only at two specific lines, of which only one is executed. To avoid confusion, we will refer to `before[x]` to mean the content of the buffer array `out[x]` before the search block executes and `after[x]` to be the state of the buffer array afterwards. To prove our loop invariant we will assume that `before[j]` $= lmp($`before[i]`$, split_j)$ at the beginning of the search loop and show that `after[j]` $= lmp($`after[i]`$, split_j)$ at the end. In one execution of the search block, one symbol will be added to `out[i]` and hence `after[i] = before[i]+r`. First, we analyze the resulting behavior when the first loop is invoked.

Case 1: If `temp.cdf > thresh[j]` then the algorithm calls `out[j].append(r)` and `after[j] = before[j]+r`. Since `temp.cdf` $= cdf($`before[i]+s`$)$ the following hold:

$$cdf(\texttt{before[i]+r}) \leq \texttt{thresh[i]} \leq \texttt{thresh[j]} < cdf(\texttt{before[i]+s})$$

$$\therefore \texttt{before[i] + r} \leq split_j < \texttt{before[i]+s}$$

Since `s` is the direct successor of `r`, we conclude that `before[i]+r` (a.k.a. `after[i]`) is a prefix of $split_j$. This fact alone suffices to prove our goal. From here, we see that:

$$\texttt{after[i]} = lmp(\texttt{after[i]}, split_j)$$

$$\texttt{before[i]} = lmp(\texttt{before[i]}, split_j) = \texttt{before[j]}$$

$$\therefore \texttt{after[j]} = \texttt{after[i]}$$

$$\therefore \texttt{after[j]} = lmp(\texttt{after[i]}, split_j)$$

Case 2: If `temp.cdf` $\leq$ `thresh[j]` then no action is performed by the algorithm on `out[j]`.

$$\texttt{after[i]} = \texttt{before[i]+r} < \texttt{before[i] + s} \leq split_j$$

Therefore `after[i]` is not a prefix of $split_j$ and this is enough to conclude:

$$\texttt{after[j]} = \texttt{before[j]} = lmp(\texttt{before[i]}, split_j) = lmp(\texttt{after[i]}, split_j)$$

Now we analyze the resulting behavior if the second loop is invoked. The second loop is invoked if no such `s` was found to make cdf(`before[i]+s`) $>$ `thresh[i]` and therefore `r` represents the highest symbol in the alphabet.

Case 3: When `before[i].length() = before[j].length()` we can also infer that `before[i]` $=$ `before[j]` and hence `before[i]` is a prefix of $split_j$. Furthermore,

$$\texttt{before[i]+r} \leq split_i \leq split_j$$

This alone suffices to show that `before[i]+r` is a prefix of $split_j$ since there is no symbol greater than `r`. The proof resembles *Case 1* at this point.

Case 4: Since `before[i].length()` $\neq$ `before[j].length()` we can further say that `before[i]` $\neq$ `before[j]`. Therefore `before[i]` is not a prefix of $split_j$ and hence `after[i]` is not a prefix either. The proof resembles *Case 2* at this point. □

The benefits of *amortization* can be large. The invariant indicates that for each character of the longest prefix between $split_i$ and `out[i-1]`, amortization eliminates one execution of the search block. Deeper in the protocol when split strings tend to cluster together, the effects can be dramatic. However, our scheme does not reach the theoretical minimum number of table queries so perhaps there is some room for improvement. We did not choose to pursue this research further for two reasons: our performance tests indicated that algorithm performance was fast enough that we felt we would achieve diminishing returns. Second, completely minimizing the number of table lookups would involve the algorithm maintaining more data and "bookkeeping" tasks which would incur additional performance overhead. It was not clear that additional complexity would actually improve performance.

5.1.5 Html Parser

In order for the server to perform the *Speculative Updating* feature, Infranet needs to be able to parse all links and image tags out of any secretly requested HTML files. HtmlParser takes a buffer of characters and if the content type is listed as "text/html", looks for any words preceded by the terms src, href, or url followed by an equal sign (allowing for whitespace in between terms). When the HtmlParser has reached the end of the document, it returns a list of the URLs it has parsed.

HtmlParser also manipulates these URLs in a variety of ways. If the html file lists a relative URL, HtmlParser prepends the server name and relative path. If the URL contains any anchor tags, they are stripped from the URL. Leading and trailing quotes need to be removed and special characters need to be converted (for example '&' becomes '&' and a space becomes '%20'). If the port is explicitly set to 80 in the URL, the port field is removed. These steps make Infranet better at predicting client behavior since these actions are taken by most major browsers.

```
public String[] split(String lower, String upper, double[] thresh) {
  if (nothingBetween(lower, upper)) return null;

  Bound bound = new Bound(lower + "\0");
  StringBuffer[] out = new StringBuffer[thresh.length];
  String given = longestPrefix(lower, upper));

  for(int i = 1; i < out.length; ++i)
    out[i] = new StringBuffer(given);

  out[0] = new StringBuffer(lower);
  for(int i = 1; i < out.length; ++i) {
    boolean notDone = true;
    while (notDone) {
      StringBuffer testRoot = new StringBuffer(out[i]);
      String r = "";
      char border = bound.getBorderChar(out[i].length());

      search: {
        for(Iterator iter = getSymbols(); iter.hasNext();) {
          String s = (String)iter.next();
          if (s.charAt(0) > border) {
            String test = testRoot.append(s).toString();
            Bound temp = bound.update(test);
            if (temp.cdf > thresh[i]) {
              for(int j = i; temp.cdf > thresh[j]; ++j) out[j].append(r);
              break search;
            }
            bound = temp;
            testRoot.deleteCharAt(testRoot.length()-1);
          }
          r = s;
        }
        for(int j = i; out[j].length()==out[i].length(); ++j) out[j].append(r);
      }
      notDone = (prev.length() > 0) && (bound.cdf <= thresh[i-1]);
    }
  }
  return out;
}
```

Figure 5-5: Implementation of the split string algorithm.

```
private static class Bound {
  private double commonCdf, userCdf;
  private String commonBound, userBound;

  public double cdf;
  public String bound;

  public Bound(String test) {
    commonCdf = COMMON_RATE * commonTable.cdf(test);
    commonBound = commonTable.bound();
    userCdf = USER_RATE * userTable.cdf(test);
    userBound = userTable.bound();
    calculateTotals();
  }

  // copy constructor
  private Bound(Bound b) {...}

  public Bound update(String test) {
    Bound b = new Bound(this);
    if (test.compareTo(b.userBound) > 0) {
      b.userCdf = USER_RATE * userTable.cdf(test);
      b.userBound = user.bound();
    }
    if (test.compareTo(b.commonBound) > 0) {
      b.commonCdf = COMMON_RATE * commonTable.cdf(test);
      b.commonBound = common.bound();
    }
    b.calculateTotals();
    return b;
  }

  public char getBorderChar(int index) {
    if (index == bound.length) return 0;
    return bound.charAt(index);
  }

  private void calculateTotals() {
    cdf = userCdf + commonCdf;
    bound = (userBound.compareTo(commonBound) > 0) ? userBound : commonBound;
  }

}
```

Figure 5-6: Implementation of the Bound class

```
Infranet::Infranet(char* file) {
  // read in host, port, default_page, initialkey, and pubkey_file from file
  ...
  server = new Server(host, port);

  Crypto::generateKey(shared_key);
  char cipher[MAX_STRING];
  Crypto::encrypt(shared_key, cipher, pubkey_file);

  mapping = new HexMap();
  sendData(cipher, initialkey);
  delete mapping;
  mapping = new RangeMap();
}

void Infranet::getRequest(char* req, char* outFile) {
  sendData(req, shared_key);
  getHiddenData(shared_key, outFile);
}
```

Figure 5-7: High level API for creating and using a covert tunnel to an Infranet responder.

5.2 Client

The client also has an object oriented software architecture implemented in C++. The classes that compose the Infranet client are in many ways analogous to their counterparts in the Infranet server. Implementation was significantly easier on the client side and did not present many challenges. Here we give an overview of the architecture and API with brief code samples.

The Infranet class is a high level object responsible for secretly transmitting HTTP requests to Infranet servers and collecting the secret responses. The constructor takes as a parameter a configuration file which contains the address, port, default page, initial key, and public key file of an Infranet server. The constructor then generates a connection to the server and negotiates the tunnel setup phase of the Infranet protocol. Subsequent calls to the method `getRequest(..)` transmit the client's secret request `req` upstream and store the secret response to the file `outFile`.

The methods `sendData` and `getHiddenResponse` perform upstream and downstream communication respectively. Note that Infranet calls `sendData` during both the tunnel initialization and upstream communication phases of the protocol, even though one phase uses a static dictionary and the other uses Range Mapping as its communication protocol. Analogous to the server side, we have implemented two classes `HexMap` and `RangeMap` which are responsible for handling the details of their respective upstream communication protocols. Since they both extend a common abstract base class, Infranet can change its protocol by merely swapping the correct class into the `mapping` variable.

The method `getHiddenData` is responsible for requesting images and extracting hidden messages from the server responses. The parameter `key` specifies which steganographic key to use in extracting the data, and outFile represents where the extracted data should be stored. The method `extractFromImage` is responsible for invoking the Outguess algorithm, parsing the downstream header from the resulting data, and appending it to the end of `outFile`. The function then checks the length and offset of the data chunk to determine whether to continue requesting images.

To determine which images to download, Infranet maintains a queue, `imageList`, implemented using the `deque` template from the Standard Template Library. Every time Infranet receives an HTML page with an image link, that URL is added to the imageList. The method `getHiddenData` pops the head of this queue to determine what URLs the client should request when it needs to receive a message from the server. If the queue is empty, the Infranet client requests the "default page", a URL that is guaranteed to contain image links. The default page is specified in the configuration file.

The `Server` class maintains the HTTP connections to the Infranet server, using the standard functions from Unix socket library (`read`, `write`, `connect`, and `accept`) to communicate. The `Server` class mainly exists to provide a convenient abstraction between the `Infranet` class and some of the messy details associated with the HTTP protocol. For example, web servers can specify the end of a response in variety of ways. They can either close the connection, set the `Content-Length` header, or send the response in a special format by specifying the header `Transfer-encoding: chunked`. The `Server` class handles each of these situations gracefully, transparently handing the response up to the higher level classes. It also transparently handles HTTP session information by parsing and setting the `Cookie` headers of all requests and responses.

The `HtmlParser` class is used to parse out image links from the HTML pages. Its implementation resembles that of its counterpart in the server software. The `Crypto` class is basically a wrapper around the OpenSSL [31] library functions, providing a convenient, high-level abstraction for the `Infranet` class to perform RSA encryption.

5.3 Additional Tools

Infranet employs a variety of open source tools and libraries. On the client side the OpenSSL library [31] was used to perform the RSA encryption. On the server side the BouncyCastle [4] cryptography library was used to perform the RSA decryptions, a plug-in to the JCE (Java Cryptographic Extensions) framework [23]. Niel Provos' Outguess [32] was used to perform steganography on both the client and server sides. The code has been compiled with gcc [13] running either Linux or Cygwin over Windows [7]. To populate our common table, we used a Web proxy trace containing 174,100 unique URLs from the Palo Alto IRCache proxy on January 27, 2002 [20]. This data was also used to determine the frequency of each character for the Uniform-Split algorithm.

```
void Infranet::sendData(char* req, char* key) {
  mapping->setSymbol(req);
  while (!mapping->done()) {
    if (mapping->timeToUpdate()) {
      getHiddenData(key, TEMP_FILE);
      mapping->setDownstream(TEMP_FILE);
      System::remove(TEMP_FILE);
    }
    char path[MAX_STRING];
    mapping->getUpstream(path);

    char response[MAX_BUF];
    int response_len = MAX_BUF-1;
    server->getPath(path, response, &response_len);

    HtmlParser::extractImages(response, path, &imageList);
  }
}
```

Figure 5-8: Upstream communication algorithm for an Infranet client. By using the Mapping abstraction, the requester can reuse the same the code to send information through a static dictionary or range mapper.

```
void Infranet::getHiddenData(char* key, char* outFile) {
  sprintf(zipFile, "%s.gz" outFile);
  bool done = 0;
  while (!done) {
    char image[MEGA_BUF];
    int image_len = MEGA_BUF;
    char* image_path = nextImage();
    server->getPath(image_path, image, &image_len);
    delete image_path;

    done = extractFromImage(image, image_len, key, zipFile);
  }
  System::gunzip(zipFile);
}
```

Figure 5-9: Function requests images, extracts steganographic data and saves output to the location specified by out_file.

Chapter 6

Security & Performance Analysis

In this chapter we discuss the types of attacks a censor might deploy and evaluate their effectiveness. For this analysis, we assume that the censor has the power to inspect and modify all traffic between Infranet clients and servers, but no control over the machines themselves. We do not make any assumptions about the computational or memory resources that are available to a censor except that it cannot break public key encryption or determine the contents of a steganographically embedded message without the corresponding secret key. We note that the censor can always thwart Infranet by shutting down its network. Therefore, our threat model assumes that the censor does have some vested interest in maintaining a connection to the Internet and wishes to allow innocuous users on their network.

The original paper on Infranet presents [10] its own security analysis of the system. We will revisit and reevaluate many of these ideas, presenting a more in depth discussion. We also introduce a new class of attacks, *Selective Degradation*, and propose possible defenses.

6.1 Discovery Attacks

A censor may attempt to discover Infranet clients and servers by looking for anomalies in HTTP requests or responses. Because of Infranet's covert nature, a censor only has limited means of attack. A client is always making a request for legitimate content. Except in the case of images, the server is always returning the same content for legitimate and Infranet clients alike. By its nature, there are only two types of anomalies that can be exploited by a censor: detecting changes in images and looking broadly at statistical properties of a browsing session, in hopes of finding timing or request patterns that are suspicious.

6.1.1 Steganographic Anomalies

A censor may attempt to detect the use of steganography by searching for duplicate image requests that produce slightly different responses. More specifically, it could maintain a hash table mapping URLs of image requests to the checksum of the corresponding image returned by the server. In the case of inconsistency, the system could flag the censor for further investigation.

We can counter this threat by avoiding duplicate requests for images. The server, when servicing HTML files can dynamically create a new URL for each reference to an image. The client then requests this URL instead of the original file name. If the server always generates a unique URL then the censor will never detect a problem.

However, a censor can thwart this strategy by taking a more active approach to discovering Infranet use. Instead of waiting for clients to issue duplicate requests, the censor can just issue its own duplicate request, spoofed to appear from the suspected user. If the responses differ, then the system can flag the censor for further investigation.

Unfortunately, there are no easy solutions to such an attack. The server could maintain a cache of past images it has serviced as well as their associated dynamic URLs. However, with the large number of images required to covertly communicate even the most basic content, this solution may become unduly burdensome for even moderately popular servers. There are, however, two factors that inherently dampen the effectiveness of such an attack. First, it is a fairly involved and expensive attack, unpractical for a large amount of traffic. Instead, the censor would probably have to use such an attack selectively, either randomly or on clients it had already suspected of illegitimate use. Second, it is a method that would undoubtedly generate many false positives. Sites which employ web cams dynamically produce new and unique images on a regular basis. Many sites have sophisticated ways of rotating advertising on their sites. The censor will certainly detect legitimate use of many of these sites as well as Infranet sessions.

6.1.2 Statistical Traffic Anomalies

Another method of attack would be for censors to try to look for traffic anomalies, patterns of requests that serve as indicators of Infranet use.

In order to determine what can be considered an anomaly, it is first necessary to define “normal” browsing behavior. For this, Feamster et al. proposed a *link-traversal probability function* defined as such:

$$p_{ij} = P(page_i \text{ is the next request} \mid page_j \text{ was the last request})$$

The server then chooses the thresholds for its split strings in such a way that the client’s behavior conforms to this function.

The experience of building Infranet has shown us practical flaws in this approach. In our implementation of Range Mapping we discovered that client behavior is highly individualized and often changing. Since each user visits a web site with different purposes, deviation from a universal link-traversal probability function is not necessarily suspicious. Additionally, we raise the question of how a censor might even be able to discover this probability function. One method would be to log the behavior of normal users and use some kind of statistical analysis to model client behavior. However, before the censor can log normal behavior, it must first be able to distinguish between normal users and Infranet users. Hence, the censor must have the exact information it is trying to obtain, which is clearly circular.

Furthermore, Infranet contains one obvious statistical anomaly that is not encapsulated in the link-traversal model of client behavior, namely the sheer number of requests that must be generated over the course of an average Infranet session. Just the initialization portion alone of the protocol takes 256 requests to transmit the encrypted shared key.

We can mitigate this problem by slowing down the rate of transmission and putting a randomized delay between requests that spreads the Infranet session over a large time frame to avoid suspicion. However, it is an undeniable fact that most web sites on the Internet do not contain enough content to justify this level of traffic. Even if spread over the course of days, several hundred requests to a web site could reasonably be perceived as suspicious to a censor.

6.2 Active Attacks

A censor may attempt to attack Infranet by manipulating traffic in a way that intentionally disrupts Infranet sessions, but leaves normal users relatively unaffected.

Infranet effectively precludes many forms of active attacks. Its design prevents the filtering of traffic based on IP address or other characteristics since Infranet servers disguise their anti-censorship functionality. Nor can a censor filter based on content since each individual Infranet server acts as a distinctive and unique web site.

A more involved attack could involve spoofing the client or server's IP address to disrupt Infranet sessions. Examples of disruptive behavior include:

1. A *mutation* attack - Changing a client's request URL on route to the server.
2. An *addition* attack - Sending a spurious request to the server, spoofing the client's IP address.
3. A *deletion* attack - Dropping a request from the client and sending back a fake acknowledgement, spoofing the server's IP address.

Analogously, the censor can perform these attacks on server responses as well. The most likely target of these attacks image files, where the censor could randomly flip bits to destroy the steganographic message.

Such attacks could certainly disrupt Infranet sessions. However, they would also damage innocuous web users if applied haphazardly. Since we assume that censors have a vested interest in allowing legitimate HTTP requests, a censors will probably take a more subtle approach. Instead a censor may attempt to *selectively* degrade performance in a manner that minimizes the inconvenience to normal web users, but maximizes the disruption of Infranet sessions.

6.2.1 Selective Degradation

As a simple example, a censor may decide to perform a deletion spoofing attack as described above with probability p correctly forward a packet with probability $1 - p$. For a normal user, such a disruption will most likely trigger the client's browser to request the missing content again, with the client only perceiving a slightly longer download time for browsing. However, such a disruption will push an Infranet client and server out of sync, forcing the client to reinitialize the entire session.

One way to quantitatively measure the inconvenience to Infranet and legitimate users is to calculate the expected number of requests necessary to fulfill a request. For a legitimate client, this expected value is $\frac{1}{1-p}$. However, let's assume for this analysis that an Infranet client needs n successive requests in the visible domain to obtain one request in the covert domain. Then for an Infranet client, the expected number of visible requests comes out to $n \cdot \frac{1}{(1-p)^n}$. As n and p increase, the inconvenience grows significantly faster for Infranet clients than for legitimate web users.

In fact, a censor can do even better with a slightly more complex randomized degradation attack. Instead of a universal drop probability p that is applied to each packet independently, consider if a router assigns a different p for each source IP address. For the first one hundred requests from a given IP address the router sets $p = 0$, ensuring no malicious attacks. At that point, the probability of a deletion attack increments by $\frac{1}{100}$. After a deletion attack on a client, p returns to 0 and the process repeats.

Let us again analyze the inconvenience of this scheme on legitimate and Infranet clients. Without doing any calculations, we observe that every deletion attack on a client follows at least 100 successful requests. We also observe that one attack must occur in every 200 requests. Therefore, between 99% and 99.5% of a legitimate client's requests will be serviced so for all practical purposes there is no inconvenience. However, every Infranet sessions will be disrupted since the initialization portion of the protocol alone takes over 200 requests.

It is also important to note that the degradation of traffic can also be used as the basis for a discovery attack. Currently when the protocol fails, the client reinitializes the covert communication channel, obtaining a new ID from the server which Infranet does not. Obtaining a new ID is not in itself a suspicious activity. It can be the result of something simple like closing and reopening a browser window. However, if a censor performs a disruption attack on a client several times and each time the client subsequently obtains a new ID, that could be construed as suspicious behavior.

Clearly, Infranet needs a graceful mechanism to recover from spurious or unexpectedly dropped requests.

6.2.2 Recovery

With the assumption that a malicious censor can intercept all traffic between an Infranet client and server, there is no direct means of confirming correct transmission. The censor can spoof fake requests and responses as well as acknowledgements. Therefore, we must augment our protocols to include mechanisms for clients to discover any errors in the upstream or downstream communication and to rollback to a previous state if necessary. The server specially designates several URLs for the purpose of signaling a rollback and transmits them to the client during the Tunnel Setup phase.

During the initialization process, the encrypted secret key is transmitted using a static dictionary. The encrypted stream of bytes is converted to hexadecimal and each digit (`0` through `F`) is assigned an associated URL in the visible domain. The client communicates each digit by requesting its associated URL. To increase the robustness of this protocol, the client can periodically ask the server for the status of the transmission by requesting an image. This indicates to the server to send to the client by downstream communication the partial string that the server has obtained so far. If the string contains any mistakes the client can always request one of the specially designated rollback URLs, signaling to the server to remove one symbol from the end of its partial string.

During downstream communication, the client can detect problems since each fragment transmitted by the server also contains a header, indicating the length of the fragment and the offset. If the offset does not match up with the number of bytes received by the client, the client infers that a fragment was lost and signals to the server to retransmit the downstream message.

During upstream communication, one of two errors can happen: either the client's request does not reach the server or the censor generates a spurious request from the client. If the request does not reach the server, the client will attempt to obtain the next set of split strings, but the server will not have any to offer. In this case, the client can try sending its previous request again. If a spurious request is generated by the censor, then the client will obtain a set of split strings, but the secret string will not fall within a range between any of them. In this case, the client requests a rollback to return to the previous iteration.

If the censor is really malicious, it may be able to block out sequences of requests from a client, thereby dropping the rollback requests as well. As a final backup, we propose adding a *soft restart* mechanism to the protocol, basically aborting the current transaction and returning back to the end of the Tunnel Setup phase with the same shared key and session ID. This avoids the weakness of reinitialization.

6.3 Performance Experiments

Over the course of this thesis, we have introduced three main innovations to the Range Mapping protocol for the purpose of improving end-to-end system performance. These are:

1. Optimizations to the Common-Split algorithm to reduce the number of redundant table lookups. This reduces the amount of time spent calculating split strings.

2. Using the Interval-Split instead of the Threshold-Split in our algorithms to reduces the amount of downstream information that must be conveyed each iteration.

3. Attempting to speculate the client's next request and updating the probability distribution accordingly to generate more accurate guesses. This reduces the number of iterations of the protocol.

We have performed a series of tests to determine whether these additions would have any significant impact on end-to-end performance over a simpler, more naive implementation. Our test environment consisted of a Pentium II 250MHz processor with 256MB of RAM running Red Hat Linux 7.1.2 as the server and a Windows 2000 machine with 1GHz processor and 512MB of RAM as our client. The client and server communicated over a 6.0Mbps wireless link. Web browsing was performed with Internet Explorer 5.5, set with the standard configuration options. Additionally, we cleared the Temporary Internet Files cache between each test.

Our test suite consisted of visiting a total of six predefined web sites through our browser. When IE finished loading each web page, we arbitrarily followed one link on each page. For one of our sites, `http://www.google.com`, instead of following a link we entered a search for the term "infranet". Note that our suite actually makes more than twelve secret requests, since IE automatically fetches image and Javascript files if necessary. This is intentional. We wanted to come as close as possible to the sequence of requests that a real user might download in a real browsing session. In all, this test suite consisted of 222 requests in the covert domain, mapping to thousands of requests in the visible domain.

A skeptic may correctly argue that the test suite we have devised is too simplistic and does not accurately reflect performance of a real system. After all, one of the central assertions of this thesis is that the client's distribution of URLs is difficult to accurately model. Therefore, there is no obvious "representative" set of URLs that can accurately predict performance. To assuage this criticism, we have run each test on the system, once with each of the six web sites added to the common table, and once where each have been explicitly removed from the common table. This represents a wide range of possible situations that Infranet may encounter in the real world. In reality, performance will probably fall somewhere in the middle.

Implementation	Secret request length	Iterations	Total download	Total time
`naive`	52.6	26.5	597.9 KB	25.97 s
`optimized`	52.8	12.0	242.9 KB	6.06 s

Table 6.1: Performance results with URLs in common table.

Implementation	Secret request length	Iterations	Total download	Total time
`naive`	52.9	36.3	589.9 KB	26.25 s
`optimized`	58.5	14.2	264.8 KB	6.50 s

Table 6.2: Performance results of implementations with URLs *removed* from the common table. The reason that the optimized version works so well is because Speculative Updating still picks up most of the URLs.

The results of our experiments are shown in Tables 6.1 and 6.2. To judge the overall performance of the system, we measured the average time to obtain a secret request. Additionally, we measured the average number of iterations taken by the Range Mapping protocol and the total number of visible bytes downloaded for each secret transaction. As a point of comparison we have shown is the average length of our secret requests, which indicates the number of iterations a static dictionary would have taken on the same set of requests. Note that our naive implementation still outperform the static dictionary when all secret URLs miss the common table.

One phenomenon of interest is the fact that there are discrepancies in request size, even as our tests remain constant. Many web sites dynamically change their images and links (usually for advertising) and therefore, a little bit of randomness is inherently built into the process. To prevent too much noise, all testing was performed over the same evening, May 5th 2003. This phenomenon further serves to highlight the importance of dynamically updating the probability distribution.

In overall, end-to-end speed, we found our system to produce a roughly 75% improvement over a naive implementation. We also saw more than a 50% drop in iterations and bytes downloaded as well. For an additional perspective, Figure 6-1 shows a scatter plot of the number of iterations taken by the protocol as a function of request length. Figure 6-2 shows the cumulative distribution of the end-to-end performance of the protocol.

Furthermore, our measurements indicate that removing URLs from our common table did not have a significantly noticeable impact on the overall performance of our optimized system. It had no effect on many of the images and dynamic links since *Speculative Updating* handled those requests in both cases. On the URLs we explicitly removed from the table, Unform-Split did a better than expected job of converging on the correct string quickly. We attribute this to the fact that web sites generally make an effort to make their main pages short and easy to type, even though their links can be unwieldy. Uniform-Split performs well on these types of URLs since it favors shorter strings containing more common letters.

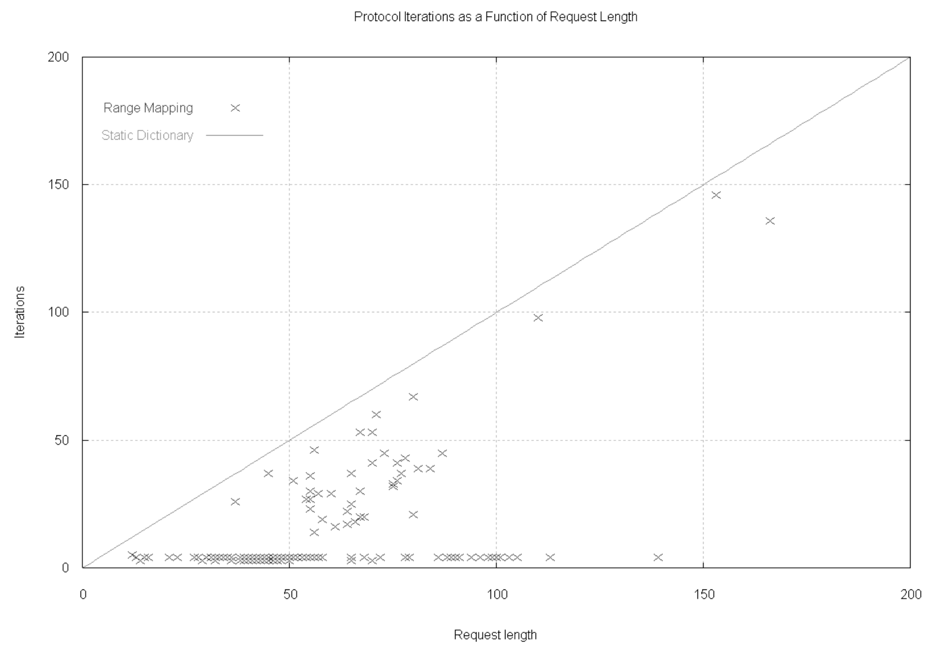

Figure 6-1: Plot of iterations as a function of request length. The dense cluster of strings along the x-axis represent secret requests successfully located by Common-Split. The remaining points represent the performance of Uniform-Split, with varying levels of success.

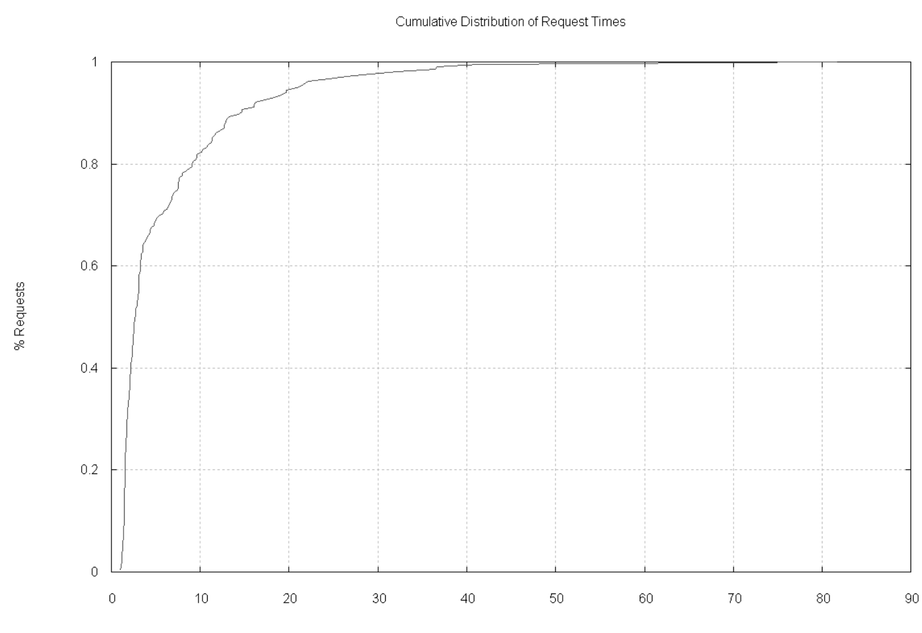

Figure 6-2: The cumulative distribution of times taken for a client to transmit the secret response and receive the secret request. Over 50% of requests are serviced within 3 seconds and over 80% are serviced with 10 seconds.

Implementation	Downstream message	Common-Table queries	Time
`naive`	1193.7 bytes	172045.2	3993.7 ms
`optimized`	312.4 bytes	358.7	29.9 ms

Table 6.3: Common-Split algorithm performance

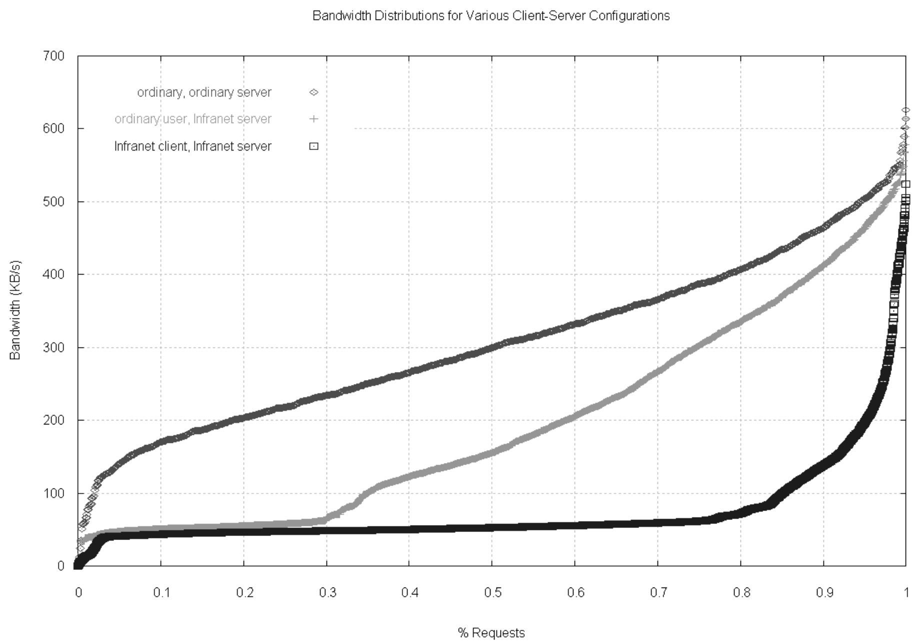

Figure 6-3: Comparison of the performance between different client-server pairings. An Infranet client communicating with an Infranet sees the lowest bandwidth on its visible requests because of the computational overhead associated with the protocol. An ordinary client communicating with an ordinary server sees the best bandwidth. The ordinary client communicating with an Infranet server sees a mix of the two.

Next, we analyzed the effects of our optimization on the performance of the split string algorithm. Table 6.3 summarizes these results. In total, we found that our algorithm met our performance expectations, significantly decreasing table lookups, message sizes, and overall times.

Finally, we decided to compare our Infranet server with an innocuous web server to see how much of a discrepancy in performance we would see. Taking a log of the visible requests recorded between Infranet clients and servers in our previous tests, we configured an innocuous client and server to make the same sequence of requests. Additionally, we configured an innocuous client to make the same requests to an Infranet server. The distribution of bandwidth for all three situations is presented in Figure 6-3. The difference in performance is noticeable, indicating that the Infranet protocol is nontrivial to compute. It indicates that there is still plenty of work to do. However, the tests were run on a server with below average computational power. We believe that the discrepancy will be smaller on systems with faster processors.

Chapter 7

Conclusion

In this thesis, we set out to discover if Infranet, and in particular the Range Mapping protocol, could feasibly be implemented in a fast, secure, and robust software system. Through this experience, we have learned many new ways to refine and improve on the original Infranet protocol and our experiments confirm the viability and effectiveness of our techniques. In this chapter we summarize the main contributions of thesis, with some additional thoughts on the possible broader applications of our work beyond the goal of circumventing censorship. Finally, we highlight the future challenges that Infranet faces.

7.1 Contributions

The original concept of Range Mapping is based on an asymmetric communication protocol first developed by Adler and Maggs [1], with refinements by Watkinson et al. [44] Their research explored the problem of minimizing upstream communication over a channel with significantly higher bandwidth in the downstream direction than upstream. The algorithms they developed exploited knowledge of the probability distribution function of the client's message to efficiently make use of the channel.

But while Adler and Maggs provided a solid theoretical foundation for Range Mapping, the practical task of building Infranet led us to appreciate the true difficulty in accurately calculating the distribution of requests. The most important contribution of this thesis is an improvement to the Range Mapping algorithm that guarantees the convergence of the protocol on the client's secret string, even for strings outside the distribution. This technique, called Uniform-Split leverages a new concept, the *Interval Split*, which is analogous to the notion of split strings developed by Adler and Maggs. It also exploits the technique called *Renormalization* to avoid floating point calculation errors and scale to arbitrarily long secret request strings.

We have also introduced new techniques to improve the speed of the Range Mapping protocol, attacking this problem from several angles. First, important algorithmic optimizations make the calculation of split strings significantly faster. We introduce a technique to avoid redundant calculations across multiple split string calculations, thereby lowering the amortized cost of each. Additionally, we exploit the discrete and step-wise nature of our probability distribution to avoid unnecessary queries to our distribution tables.

In order to reduce downstream communication, the system communicates its split string in a syntax designed to reduce message size. Finally, it generates a more accurate probability distribution function of secret request strings by attempting to guess the client's future behavior by analyzing the results of its previous secret requests, a technique we have entitled *Speculative Updating*. A more accurate distribution function leads the Range Mapping protocol to converge on the correct secret string in less iterations.

In the area of security, we have proposed additional features to our protocol that allow the clients and servers to undo mistakes and rollback to previous states in the case of errors. This resiliency reduces the ability of censors to intentionally degrade the quality of service in its network in an attempt to cripple Infranet communication but leave innocuous users only mildly inconvenienced.

Beyond Infranet, we envision that these ideas have practical applications to a variety of network systems. The original motivation for Adler and Maggs' work is based on one author's attempt to set up a home network with a 2Mbps wireless ethernet link for downloads and a 28.8Kbps modem for uploads. Should asymmetric communication networks become prevalent, a protocol such as Range Mapping may be an effective way to optimize HTTP traffic. Another possible application of our research is in systems that need to fetch web pages but require extremely low latency. If memory and storage are relatively inexpensive, a strategy of speculating future requests and caching their results could be a viable idea for something like a Squid Proxy Cache [39].

7.2 Future Work

Though we have greatly increased our understanding of the complexities of implementing Infranet, there still remains room for improvement. One problem which we have noticed is that the layout of the Infranet server's web site influences the effectiveness of the protocol. In certain cases, glaring statistical anomalies can arise. Consider the situation where a couple of pages are heavily populated with images and elsewhere the distribution is sparse. Since the Range Mapping protocol relies on downstream messages at each iteration, client will need to consistently return to these pages when a downstream message is needed. One interesting future research project would be to explore the possibility of a hybrid implicit/dictionary/range scheme where the client uses Range Mapping when images are available but can still somehow transmit information to the server otherwise.

We also believe that there is a little more performance can be squeezed out of our split string algorithms. In this thesis, we focused on reducing the number of table queries. However, one optimization we did not consider is increasing the speed of these lookups. If our algorithm could remember the index in the table of previous lookups, it could restrict its future searches to a subset of entire table. This might not be a huge performance boost since lookup times grow logarithmically with the size of the table. However, it is a promising idea that we are currently exploring.

Another improvement to the Infranet protocol would be to devise a more efficient downstream message format. Currently, we use the syntax described in Section 4.6 to initially create the message and then we compress this string with gzip [16]. Even better, though, would be to devise our own encoding scheme that addresses the specific needs Infranet. We would also like to explore changes to the tunnel initialization process. Since the RSA encryption creates a 128 byte block of data, it currently takes 256 requests to transmit this cipher. This heavy duty setup protocol is suspicious, and makes Infranet more vulnerable to *selective degradation* attacks. Different encryption techniques should be explored in the future [29].

Finally, one of the largest bottlenecks in our system is the steganographic tools, both in the computational time of embedding and extracting and in the amount of information that can be placed in images. Faster processor speeds and optimizations can probably improve the first problem. However, it is unclear what the future holds for the second problem. Even as steganography becomes more sophisticated, we are seeing detection tools improve as well. In such a dynamic field, the balance of power fluctuates often. A recent paper by Fridrich et al. has successfully attacked Outguess [12].

We have recently released the source code of Infranet for public evaluation [18]. We envision the focus of our work evolving in time as we see how users and censorship organizations respond. Even though schemes like Triangle Boy [41] may be the popular anti-censorship systems of today, we believe that they contain fundamental weaknesses that censors will attack in the future. For this reason, Infranet remains a vitally important idea.

Bibliography

[1] M. Adler and B. Maggs. Protocols for asymmetric communication channels. *39th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1998.

[2] Anonymizer.com. `http://www.anonymizer.com/`.

[3] Ray Arachelian. White Noise Storm. `ftp://ftp.csua.berkeley.edu/pub/cypherpunks/steganography/wns210.zip`.

[4] Legion of the Bouncy Castle. `http://www.bouncycastle.org/`.

[5] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, February 1981.

[6] I. Clarke, O. Sandbert, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[7] Cygwin. `http://www.cygwin.com`.

[8] R. Dingledine, M. Freedman, and D. Molnar. The Free Haven project: Distributed anonymous storage service. *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, July 2002.

[9] Digital Millenium Copyright Act. `http://loc.gov/copyright/legislation/dmca.pdf`.

[10] Nick Feamster, Magdalena Balazinska, Greg Harfst, Hari Balakrishnan, and David Karger. Infranet: Circumventing censorship and surveillance. *11th USENIX Security Symposium*, August 2002.

[11] Nick Feamster, Magdalena Balazinska, Winston Wang, Hari Balakrishnan, and David Karger. Thwarting web censorship with untrusted messenger discovery. *3rd Workshop on Privacy Enhancing Technologies*, March 2003.

[12] J. Fridrich, M. Goljan, and D. Hogea. Attacking the Outguess. *Proceedings of the ACM Workshop on Multimedia and Security*, December 6 2002.

[13] GNU Compiler Collection. http://gcc.gnu.org/.

[14] Ian Goldberg, David Wagner, and Eric Brewer. Privacy-enhancing technologies for the Internet. *COMPCON '97*, February 1997.

[15] Independent JPEG Group. Jpeg-jsteg v4. ftp://ftp.funet.fi/pub/crypt/steganography.

[16] Gzip compression. http://www.gzip.org.

[17] Henry Hastur. MandelSteg. ftp://ftp.csua.berkeley.edu/pub/cypherpunks/steganography/MandelSteg1.0%.tar.Z.

[18] Infranet source code. http://www.sourceforge.net/projects/infranet/.

[19] iPlanet web server. http://www.iplanet.com/.

[20] IRCache. http://www.ircache.net/.

[21] Java 2, Enterprise Edition. http://java.sun.com/j2ee/.

[22] Java Standard Development Kit. http://java.sun.com.

[23] Java Cryptography Extension. http://java.sun.com/products/jce/.

[24] Jack Kelley. Terror groups hide behind web encryption. *USA TODAY*, Febuary 5, 2001. http://www.usatoday.com/tech/news/2001-02-05-binladen.htm.

[25] J. Lee. Companies compete to provide Saudi Internet veil. *New York Times*, November 19, 2001. http://www.nytimes.com/2001/11/19/technology/19SAUD.html.

[26] Colin Maroney. Hide and seek v4.1. ftp://ftp.csua.berkeley.edu/pub/cypherpunks/steganography/hdsk41b.zip.

[27] David McGuire. Privacy ruling goes against Verizon. *Washington Post*, April 24, 2003. http://www.washingtonpost.com/wp-dyn/articles/A33972-2003Apr24.html.

[28] P. Meller. Europe moving toward ban on Internet hate speech. *New York Times*, November 10, 2001. http://www.nytimes.com/2001/11/10/technology/10CYBE.html.

[29] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems.* Kluwer Academic Publishers, Boston, MA, 1993.

[30] The Cult of the Dead Cow (cDc). Peekabooty. http://www-peek-a-booty.org.

[31] OpenSSL. http://www.openssl.org/.

[32] Outguess. http://www.outguess.org/.

[33] USA Patriot Act (HR3162). `http://www.fincen.gov/hr3162.pdf`.

[34] N. Provos and P. Honeyman. Detecting steganographic content on the Internet. *Proceedings ISOC NDSS'02*, February 2002.

[35] Niels Provos. Defending against statistical steganalysis. *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[36] M. Reiter and A. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1:66–92, November 1998. http://www.research.att.com/projects/crowds/.

[37] R. Rivest, A. Shamir, and L. Adleman. On digital signatures and public key cryptosystems. Technical Report MIT/LCS/TR-212, MIT Laboratory of Computer Science, January 1979.

[38] RSA Data Security. The RC4 encryption algorithm. March 1992.

[39] Squid Web Proxy Cache. `http://www.squid-cache.org/`.

[40] Apache Tomcat. `http://jakarta.apache.org/tomcat/`.

[41] Triangle Boy. `http://fugu.safeweb.com/sjws/solutions/triangle_boy.html`.

[42] Voice of America. `http://www.voa.gov/`.

[43] M. Waldman and D. Mazieres. Tangler: A censorship resistant publishing system based on document entanglement. *Proceedings of the 8th ACM Conference on Computer and Communication Security*, November 2001.

[44] John Watkinson, Micah Adler, and Faith E. Fich. New protocols for asymmetric communication channels. *Proceedings of 8th International Colloquium on Structural Information and Communication Complexity*, 2001.

[45] IBM's WebSphere. `http://www-3.ibm.com/software/info1/websphere/index.jsp`.

[46] Jonathon Zittrain and Benjamin Edelmain. Empirical analysis of internet filtering in china. *IEEE Internet Computing*, 7(2):70–77, 2003.