Infranet: Circumventing Web Censorship and Surveillance

Anonymous

Abstract

This paper describes the design and implementation of Infranet*, a system that enables users to circumvent Web censorship and surveillance. An increasing number of countries and companies routinely block or monitor access to parts of the Internet. To counteract these measures, we propose a system for clients to surreptitiously retrieve sensitive content from cooperating Infranet servers. The servers provide clients access to censored sites while continuing to host normal content. This system requires carefully designed protocols that provide reasonable bandwidth without arousing suspicion. Accordingly, clients send covert messages to Infranet servers by assigning hidden meaning to standard HTTP requests. In turn, servers respond to clients by embedding data in the openly requested content. To evaluate this system and its protocols, we designed and implemented a prototype client and server. Using this evaluation, we assess performance, security, and design tradeoffs.*

1. Introduction

Many people depend on the World Wide Web to freely access electronic media from any computer in the world. However, many countries, political regimes, and corporations are attempting to restrict and monitor access to parts of the Internet.

Several countries filter Internet content at their borders. For example, China restricts access to some news sites that have been critical of the country's domestic policies. Similarly, Germany censors all Nazi-related material, and Australia's laws place a ban on pornography. In addition, Internet censorship repeatedly threatens to cross organizational boundaries. For example, the Supreme Court recently rejected France's request to censor Nazi-related material on Yahoo's site [7]. Separately, the Council of Europe is attempting to ban racist and hate speech from the Internet by extending its cybercrime convention [4].

Censorship and surveillance also extend into free enterprise, with companies commonly blocking access to sites that are not related to conducting business. In addition to blocking sites, many companies routinely monitor their employees' Internet usage.

This paper focuses on the technical problems of circumventing Web censorship and largely ignores the many related political, legal, and policy issues. In particular, we examine how a user with partial Internet access can view restricted content and maintain plausible deniability.

Many organizations control Internet traffic in an attempt to permit some Internet access, while censoring unwanted content. We investigate how to leverage communication with accessible servers in order to surreptitiously provide censored content. To do this, we use a concept called a *covert channel*, defined as a communication channel that is not designed or intended to transmit information at all [6]. In addition, we require the covert channel to be innocuous and secure.

We propose *Infranet*, a system that enables users to circumvent Web censorship and surveillance by establishing covert channels between clients and accessible Infranet Web servers. Using Infranet, a client requests and retrieves content from restricted sites by making a series of innocuous requests to accessible Web servers. The traffic generated by an Infranet client should be indistinguishable from ordinary Web traffic.

While both covert channels and steganography are well understood concepts, their application to covert Web browsing is new and challenging. The novelty arises from characteristics of Web traffic and constraints imposed by HTTP. Web browsing has asymmetric bandwidth requirements: much more data is downloaded from the server than is sent to the server. Additionally, HTTP requests conform to a regular format that is not amenable to arbitrary modification. The combination of these factors makes the steganographic bandwidth from client to server very small.

To avoid arousing suspicion, Infranet clients send messages using a sequence of standard HTTP requests. More specifically, a client and server share a secret dictionary that maps URLs to message fragments. A client composes messages by sending a sequence of HTTP requests that map to fragments in the dictionary. The server reassembles these fragments using the shared dictionary. To reduce the number of requests needed to assemble a message, some dictionary entries map to probable message completions. The server keeps these completions accurate by continually updating the dictionary. These updates also help maintain innocuous traffic patterns and preserve message security.

In downstream communication, the server embeds the secretly requested content in requested images. This presents several challenges. First, we want to maximize the amount of covertly requested data embedded in each requested image. Second, a censor should not be able to alter, recover, or remove the hidden content. Finally, it should be sufficiently difficult to determine which images contain data secretly requested by the client.

To assess the feasibility of this design, we implemented a prototype Infranet client and server and conducted a series of tests based on expected usage patterns to assess the performance of our system. We find that an Infranet client typically achieves between 3 and 5 kB/s of downstream bandwidth, while maintaining innocuous browsing patterns. For disk bound workloads, an Infranet server performs comparably to a regular Apache server.

In Section 2, we describe the problem in further detail and outline the capabilities of our adversary. Section 3 describes the system design, including important design choices. We describe our implementation in Section 4 and discuss the security of our protocol in Section 5. We evaluate the performance of our prototype in Section 6. We discuss related projects in Section 7. Finally, we conclude in Section 8.

2. Problem description

In this section, we outline the design goals for Infranet and describe the threats that our system defends against.

2.1 Requirements

Infranet's overriding purpose is to provide covert access to sensitive Web content. Secret communication, using steganography or covert channels, is well studied. Still, we must carefully apply these techniques to this problem. Namely, the system must maximize usability while minimizing the chance of discovery.

A useful system for circumventing Internet censorship must have the following properties:

- **Accessibility:** An Infranet user must have access to the same content as an uncensored user.
- **Deniability:** Network traffic generated by Infranet browsing must be indistinguishable from normal browsing.
- **Fault tolerance:** Losing access to one Infranet server should not prevent a user from switching to another server.
- **Compatibility:** Infranet cannot require extensive modification to existing client software. For example, browser upgrades are acceptable, but kernel upgrades are undesirable. This will help ensure that the system is easily deployed and widely used.
- **Performance:** A user should browse Infranet content in the same way she normally browses the Web. Additionally, a Web server should not incur a significant performance penalty for running an Infranet server.

2.2 Threat Model

To provide a framework for Infranet's requirements, we outline our assumptions about the adversary's capabilities and limitations.

As depicted in Figure 1, the censor sits between the user and the desired content and can freely perform passive observation of network packets and HTTP requests. In addition, the censor can run a caching proxy. Active threats include reordering requests and blocking IP addresses or DNS names. A censor might also replay HTTP request traces or establish Infranet clients and servers.

However, the intermediary seeks to provide normal access to unrestricted content. Thus, we assume censors do not block ordinary features of HTTP, such as cookies. Further, we assume caching proxies respect the `pragma: no-cache` header set by Web servers. Finally, we assume that a censor has limited computational power and data storage. More specifically, we reject the possibility of logging and examining every data packet for any substantial time frame.

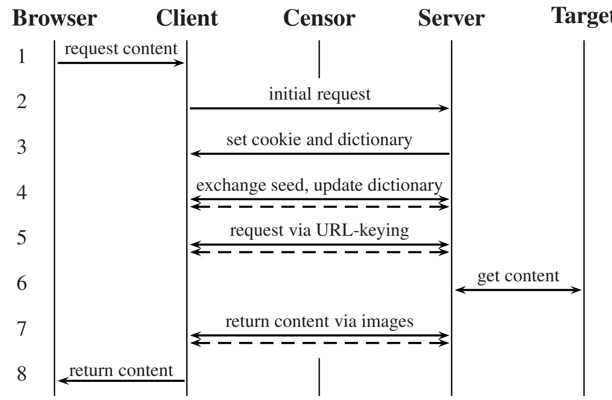


Figure 1. Summary of a complete infranet request from a user's Web browser, through the Infranet client, to an Infranet server and target server. The dashed lines indicate the possibility of multiple requests for one task.

3. Design

Having described the problem, we now present the design of Infranet's communication protocol. Figure 1 illustrates the sequence of messages required for a Web browser to obtain content from Infranet. A user retrieves content via the following steps:

1. The user's browser sends an HTTP request to the Infranet client proxy.
2. If the proxy determines that the request should be served from the Infranet, it begins communicating with an Infranet server. Communication with an Infranet server begins with a simple request for an HTML page on that server.
3. The server responds with the requested page and assigns a cookie that allows the server to keep track of the client's state. In addition, the client and server use the embedded links from the requested page to construct a communication dictionary, which associates URLs with covert message fragments.
4. The proxy then sends a random seed to the server via a sequence of HTTP requests contained within the initial dictionary. The client encrypts the seed with the server's public key. The exchange of the seed allows the client and server to generate a shared secret key, which is used to encode subsequent communication. After the key exchange, the Infranet server sends an updated version of the dictionary, encoded with the shared secret.
5. The Infranet client sends a request for censored content, using the shared dictionary as a codebook for generating the request.

6. After decoding and interpreting the hidden request, the server obtains the requested content from the target server.
7. The proxy continues to browse the Infranet server's pages. The server embeds segments of the hidden content into images as it serves them.
8. The Infranet client recovers the covert content from these images, assembles the data, and returns it to the browser.

In the following subsections we elaborate on these steps. We first describe how the client and server exchange the shared secret. We then discuss how a client sends messages to an Infranet server and how the server parses those messages. Finally, we discuss how the server returns the corresponding hidden data to the client.

3.1 Upstream Communication

An Infranet client engages in two types of upstream communication: key exchange and requests for hidden content. For both types of messages, the client uses a scheme we call *URL-keying*.

An Infranet client sends a message by requesting a sequence of URLs from the server, each of which maps to a message fragment. A message fragment could be anything from a single bit to a character string.

The mapping between URLs and message fragments is maintained by a secret dictionary, which must support three operations: insert, getFrag, and getURL. The insert procedure accepts a URL and a message fragment, and inserts a mapping in the dictionary. The assignment replaces any previous definition. The getFrag function returns the message fragment associated with a URL. The getURL procedure returns the dictionary's longest prefix match of the message, along with the URL associated with that prefix.

A client sends a message by requesting the URLs returned by repeated calls to getURL. On each iteration, the input parameter to getURL should be the remaining unsent message segment. The server will then use the getFrag method to recover the covert message.

For example, suppose the client wants to request `www.mit.edu`. Further, assume its dictionary maps `/people.html` to "www." and `/home.html` to "mit.edu". On the first iteration, the proxy would call getFrag on "www.mit.edu" and obtain `people.html`, which maps to the longest prefix "www.". Then, after requesting `/people.html` from the server, the proxy would call getFrag on "mit.edu". Finally, the proxy requests `/home.html` from the server.

3.1.1 Key Exchange

To secure communication between an Infranet client and server, the client establishes a shared secret with the server. The proxy and client then encode all future communication using the shared key. The key is established using the following protocol:

1. The proxy generates a random seed s and encrypts it as $E(PK, s)$ using the server's public key, PK. Note that PK might need to be obtained out-of-band.
2. The proxy requests a known HTML document, such as `index.html`, from the server.
3. The server responds with the document and sets a cookie, `ClientToken`, which uniquely identifies the client on subsequent requests.
4. The client and server establish an initial dictionary from the embedded links within the requested HTML document.
5. The client transmits the encrypted seed, $E(PK, s)$ using the URL-keying mechanism.
6. The server retrieves s using its private key.

3.1.2 Requesting Content

After the client and server establish a shared secret, the client can begin sending requests for Infranet content using the URL-keying procedure. Several enhancements to dictionary usage will help improve performance, make the traffic look innocuous, and keep the hidden content secure.

To increase upstream bandwidth, we incorporate URL completion,which is similar that in the navigation bar of popular Web browsers and tab-completion in many UNIX shells. As the Infranet client begins sending a request, the server sends dictionary updates, which include possible completions to the client's message. Additionally, we initialize the guesses to commonly requested strings, such as `http://` and `http://www.google.com`. Effective completions must be continually updated as the Infranet client browses the server. The server can embed these into images that are contained in pages that the client requests during URL-keying. For example, if the client has sent a partial message "ama", the server might include "zon.com" as a completion suggestion in the dictionary.

Dictionary updates also help ensure innocuous request patterns. In particular, the probability of requesting a particular URL should correspond to the probability of sending the associated message piece. Conditional probabilities for requesting URLs change as the client moves through the site; the dictionary should re-map its entries according to these changes.

Additionally, dictionary updates can prevent frequency analysis attacks. Without updates, it might be possible for an adversary to mount a known plaintext attack using frequency analysis. For example, given a static mapping and knowledge that a user might type `cnn.com`, an adversary might be able to decode the request without knowing the mapping, simply based on the Infranet client's browsing patterns. In addition to dictionary updates, effective completions can also prevent frequency analysis attacks.

3.2 Downstream Communication

The server returns data to an Infranet client in explicitly requested content. A variety of techniques exist for data insertion, including steganography and watermarking. Each of them has different properties, but all impose limitations on the amount of hidden data that can be embedded in the cover data. Hence, hidden messages will often be split up across several explicitly requested objects.

One publicly available steganography tool, `outguess` [9], hides data in images. The processed images appear unmodified under many tests. In addition, `outguess` accepts a key that is required to both encode and decode images. An Infranet server must embed data in every image that it serves. This helps ensure plausible deniablity, since every client, both Infranet and innocuous, will retrieve modified images.

Steganographic tools allow a message to be undetectably inserted into a single image. However, the resulting image changes when the embedded content changes. Because an Infranet server

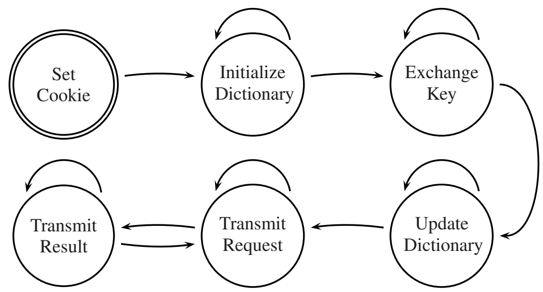


Figure 2. An illustration of the finite state machine that drives the communication between an Infranet client and server.

repeatedly embeds different messages into the same image, a censor may be able to detect and destroy embedded content more easily. Thus, we are not certain that steganography meets our design requirements.

3.3 Distributing Server Functionality

In many cases it may be desirable to distribute an Infranet server's tasks across several peering machines. Changing to another peer might help maintain normal browsing patterns. A server might not be willing to knowingly host certain content, such as pornography. Hence, a server might agree to serve encrypted files, but not receive requests directly from clients.

While we have not yet incorporated this functionality, we believe that it is an important aspect of a complete system. Giving the client the ability to make a request at one server and retrieve the associated content from a second server could potentially improve the client's plausible deniability and balance load across servers.

However, this mechanism presents significant challenges. Allowing the client to move from server to server implies that a censor could establish a client, quickly learn about many servers, and block access to them. How to distribute server functionality across peering servers is the subject of future work.

4. Implementation

The *Infranet* prototype consists of two main components: a client proxy and a server module. The *client* functions as a Web proxy and is responsible for translating a user's request for hidden content into a series of HTTP requests to Infranet servers. The client then decodes an Infranet server's responses and passes the hidden content back to the user. The *server module* translates the client's apparently innocuous messages into requests for hidden content, which is returned to the client via steganographic images. The following sections elaborate on the implementation of these two components.

4.1 Finite State Machine

The client and server operate using the finite state machine (FSM) shown in Figure 2. The FSM follows the protocol described in Section 3.

When a client first makes a request to an Infranet server, the client is in the *Set Cookie* state. This state sets the cookie, `Client-Token`, which contains a unique client ID that the server uses to track a particular client across subsequent requests. The client and server also use the requested HTML to initialize the URL-keying dictionary (state *Initialize Dictionary*). This leads to the *Exchange Key* state, where the client and server establish a shared secret, as described in Section *3.1.1*.

Once a shared secret is established, the client and server enter the *Update Dictionary* state, where the server sends the client updates to the URL-keying dictionary. With the dictionary updated, the client and server enter the *Transmit Request* state, where the client employs the dictionary to generate a series of requests to the Infranet server that secretly communicate a request for a URL. When the client sends an END character by URL-keying, the client and server enter the *Transmit Result* state, where the server fetches the requested URL and embeds the content in images requested by the client. When the fetch is complete, the server sends an EOF character to the client, at which point the client and server both re-enter the *Transmit Request* state. In the future, we plan to modify the FSM so that the client and server can periodically re-enter the *Update Dictionary* state.

4.2 Client

We implemented the Infranet client as an asynchronous Web proxy in about 2,000 lines of C++ code. We used `libtcl` for the asynchronous event-driven functionality.

When the client proxy is launched, it must know how to contact at least one Infranet server. Upon receiving a request from a Web browser, the client selects a random Infranet server that is not currently serving a request for that client. If all known servers are busy, the client queues requests until one becomes available.

After selecting a server, the client enters the FSM described in Section 4.1 and initiates communication. If this is the first contact with the server, the client starts at the *Set Cookie* state. Otherwise, the client proceeds to the *Transmit Request* state.

In the fetch state, the client extracts hidden content from every requested image. The set of embedded images on a single page is often too small to hold the entire hidden response. Therefore, the client must continue requesting Web pages and their embedded images until it receives all of the content. When receiving embedded data, the client assembles the fragments into a buffer until it receives an EOF character. The client decompresses the content and passes the result to the user's browser. If the content is an image, it is not compressed. Therefore, the client can stream portions of the hidden data as it arrives. When the request is completed, the client preserves the state associated with a server as a persistent object. On subsequent requests, the client can immediately begin transmitting requests.

4.3 Server Module

We implemented the Infranet server as an Apache module, `mod_infranet` in about 2,000 lines of C++, which we integrated with Apache 1.3.22 running on Linux 2.4.2. The Apache request cycle consists of many phases, including URI translation, content-handling, authorization, etc [13]. Our `mod_infranet` module augments the *content handler* phase of the Apache request loop. The server processes requests as it normally would but also interprets them as coded information.

4.3.1 Server Flow

We modified the content-handling phase to behave in the following fashion for every request:

1. Check to see if the client has a cookie. If so, read the state for that client from a file on disk. If not, set the client cookie and create a new file for storing that client's state.

2. Read the latest version of the dictionary for that client from a file.

3. Depending on the state of the client, perform the appropriate operation, such as adding letters to the URL-key request, embedding hidden content in the returned data, etc.

4. Write the modified client state back to the file.

Maintaining client state and returning data to the client are complicated issues that we discuss in the following sections.

4.3.2 Maintaining Client State

An important aspect of the Infranet server is maintaining client state. At any given time, the client state contains the client's ID, the FSM state that the client is in, the current dictionary content, and detailed information on pending message exchanges. Each client has a file on the server that stores its state; therefore, when the client revisits a server and presents its `ClientToken` cookie, the server retrieves the client state from the appropriate file.

Because the Apache Web server is multithreaded, maintaining consistent client state across multiple processes requires a strict file locking discipline. In particular, because many pages contain embedded links, a particular client may make multiple requests to a Web server. Because each request might correspond to an Infranet transaction, the server must make sure that each server process sees consistent client state. For example, two requests could arrive, and both server processes might think that the client was downloading hidden content from a particular byte offset. If the second process reads the client state before the first process is finished updating the state, then both will return the same portion of the hidden data to the client. To prevent this, we use aggressive file locking on the client state file. A server process obtains a lock on the file when it reads the file and holds the lock until the updated client state has been written. Therefore, two closely spaced processes will be guaranteed to read consistent client state, because the second process waits to read client state until the first is done reading.

4.3.3 Embedding Hidden Content

The server uses the steganographic tool `outguess` [9] to embed content in images and send it back to the user. Upon receiving a request, the server determines based on the content-type of the requested data whether it can embed hidden content in the response. Currently, `mod_infranet` only embeds data in JPEG images. If the server determines that it is capable of hiding information in the requested data, it uses `outguess` to embed the hidden information using the shared steganographic key. To reduce the amount of data that the server must send to the client, the server compresses HTML files with `gzip` [5] before embedding them into images.

When returning hidden data, the server creates a temporary copy of the requested image and embeds a portion of the hidden content. The server then returns this modified file to the client. This redirection is completely transparent to the client, who only knows that it received some data in response to an HTTP request, not the location of the specific file that is returned.

4.4 Cryptography

The client generates a random 20 byte seed by reading content from `/dev/random`. The seed is encrypted using the Rabin public key cryptosystem [18]. We use the ARC4 [15] stream cipher to generate a 128 bit key from the seed transmitted by the client. This key is then used with `outguess` to hide content inside images. All encryption operations are implemented with the `Crypto++` library [2].

5. Security Analysis

In this section, we discuss the security of *Infranet* in the face of an adversary with the capabilities described in Section 2. We are primarily concerned with maintaining deniability. Additionally, we seek to maintain confidentiality and authenticity of messages that pass between the client and server.

5.1 HTTP Trace and Traffic Analysis

We assume that an adversary may observe all traffic exchanged between a client and server. Furthermore, we assume that an adversary can examine a suspected Infranet server's Web logs to detect anomalous browsing behavior.

In the face of HTTP analysis, our system's biggest potential vulnerability is the deviation of an Infranet client's browsing patterns from those of a typical visit to that server. For example, a client is not likely to load slide 6 of a Powerpoint presentation immediately following a request for photo 4 in a user's photo album, but an Infranet client could potentially take such a path. To remain innocuous, Infranet client browsing patterns should correspond to the conditional probabilities of visiting a particular page given the last page requested on that site.

By appropriately constructing and adapting the dictionary that the client uses to send messages, an Infranet server can help defend against these types of attacks. An effective dictionary should have two properties: a many-to-one mapping of URLs to message fragments, and a dynamic remapping of dictionary entries.

A many-to-one dictionary mapping increases client deniability, because it gives the client a large degree of flexibility in making completions. As an extreme example, if a dictionary mapped all URLs to one of two entries, the client could exhibit normal browsing patterns while sending a message to the server. This increase in deniability comes at the expense of reduced bandwidth. We examine this problem in greater detail in Section 6.

Dynamic remapping of dictionary entries can increase deniability by associating popular next-hop URLs with likely message completions, effectively making browsing look more innocuous. Adequately mapping completions also aids deniability by reducing the number of documents that the client must request to send a message to the server.

5.2 Active Traffic Modification

We assume that an adversary can actively modify the data that is being passed between an Infranet client and server. Such modification might involve altering the bits in images that are returned to the client or inserting URLs on a Web page in an attempt to disrupt communication between client and server. Additionally, an adversary might have the ability to reorder the client's HTTP requests en route from client to server.

In general, our system is not vulnerable to modification of returned HTML pages, such as the insertion of links. There is one exception to this—before the client and server can exchange a dictionary under a shared secret, the shared secret must be established with a dictionary constructed from the links that the client browses on that site. Insertion or deletion of links on one of these pages could result in the client and server having inconsistent views of the dictionary and thus prevent key exchange.

We use the steganographic tool `outguess` [9] to defend against any modification of images that might destroy hidden data. As mentioned in Section 3.2, steganographic tools such as `outguess` may have limited utility against an active adversary when steganography is repeatedly applied to the same image. We are not entirely certain about the robustness properties of `outguess`. However, a server can probably defend against active attempts to remove hidden data by using error correcting codes or some type of watermarking technique.

Reordering of HTTP requests between client and server can potentially disrupt parts of the protocol, such as the key exchange. Data retrieval, however, is not vulnerable to a request reordering attack. During retrieval, the client makes arbitrary requests to the server, and the server embeds the data that the client requested into whatever the server returns to the client.

The hidden data that is returned in pieces to the client is potentially vulnerable to a reordering of HTTP responses, because the hidden data might be assembled out of order if the client receives responses out of order from the server. Our system is not vulnerable to this attack because it waits to hear a response from the server before making subsequent HTTP requests (i.e., our client does not make requests in parallel, thus responses cannot be reordered). However, even in the absence of this constraint, if each portion of hidden data is prepended with a small header that indicates its offset in the hidden data, a client will still be able to reassemble portions of the hidden data, even if various portions are received out of order.

5.3 Joining Infranet as Client or Server

It is reasonable to assume that an adversary may attempt to discover Infranet users by establishing an Infranet client and server. As an Infranet client, a censor may be able to learn the identities of other Infranet servers and begin blocking access to those servers. This becomes a more valid concern when server functionality is distributed, as described in Section 3.3. An adversary might try to monitor a key exchange between the client and server, establish a client, and replay the same encypted seed to the server at a later point. However, because the adversary has no way of recovering the original seed, it cannot derive the shared secret which was generated during that key exchange.

Alternatively, an adversary could join *Infranet* as a server. If an adversary joins Infranet as a server, it may be able to determine whether the client is an Infranet client or an innocent client based on whether the client's requests decode to any meaningful requests for hidden data. A client may defend against this by changing Infranet servers frequently; however, this still allows colluding servers to target clients. A better solution requires a valid dictionary to have a certain number of URLs that map to any given entry in the dictionary. A client that has a many-to-one dictionary mapping has much more control over its browsing patterns and can remain innocuous much more easily.

	Comp. Ratio	Time (μsec)	Bandwidth (kB/sec)
Average	0.35	4343.51	4592.61
Best	0.16	1892	7662.33
Worst	0.74	9992	259.00

Table 1. Gzip performance on the HTML index pages from 20 popular Web sites over 10 independent trials. `gzip` **can reduce HTML files to as much as 16% of their original size, and runs in < 10 milliseconds.** `gzip` **does not appear to be a performance bottleneck.**

5.4 Filtering

An adversary has the potential to block access to certain sites based on IP address or DNS name. Of course, if a censor blocks access to a machine, the client will not be able to access it. Thus, it is essential that Infranet is both accessible on prominent sites that a censor would rather not block and widely distributed across a large number of servers. These requirements imply that a censor would not be able to block access to Infranet without blocking access to a large portion of the Internet.

We assume that an adversary might gradually learn about servers that are running Infranet and block access to these sites (particularly if they are not prominent Web sites). Therefore, the system must limit the rate at which Infranet nodes are discovered. We currently have no mechanism in place to maintain server deniability; this is an area of future work.

6. Performance Evaluation

In this section, we examine the performance of the Infranet client and server. We study the overhead imposed on a Web server by running the Infranet module and compare end-to-end performance of Infranet to normal Web browsing. We also present the performance breakdown by state. Additionally, we study how certain optimizations, such as intelligent dictionary construction, compressing HTML, and changing the size of images on the Infranet server can improve performance.

All of our performance tests were run using Apache 1.3.22 with `mod_infranet` on a 1.8 GHz Pentium 4 with 1 GB of RAM. For many performance tests, we also ran the Infranet client and a Perl script that emulates a browser, from the same machine.

6.1 Microbenchmarks

In order to reason better about the performance implications of various system components, we performed several microbenchmarks. First, we ran a microbenchmark of `outguess` in an attempt to determine the rate at which it can embed data into images.

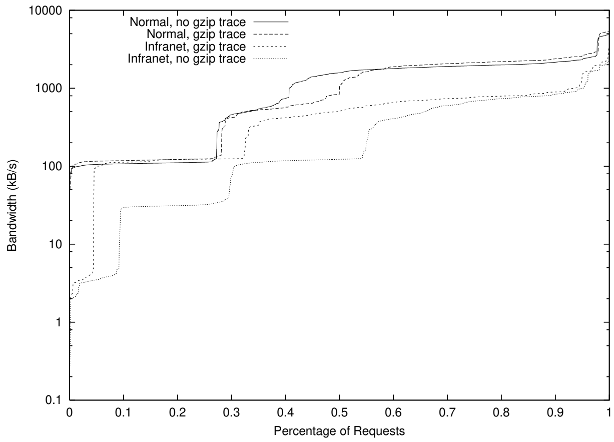


Figure 3. The effect of running Infranet on Web server bandwidth. `gzip` compression alleviates the extra overhead incurred by running an Infranet server. For disk bound workloads, an Infranet server performs comparably to a regular Apache server.

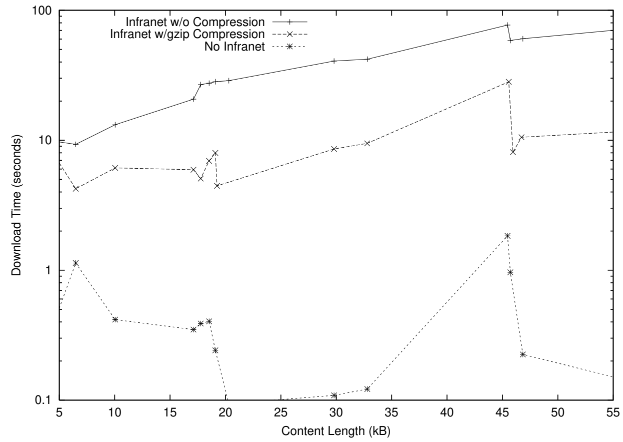


Figure 4. End-to-end download times vs. content-length, for regular Web use, and for Infranet with and without `gzip` compression of embedded data. `gzip` compression increases the bandwidth of Infranet by roughly a factor of 10. Even with compression, however, Infranet's bandwidth is roughly 10% of normal Web browsing. `gzip`-ing of images provides negligible performance gains.

Our measurements indicate that `outguess` embeds data into an image at a rate of 20 kB/sec, and that the time that `outguess` takes to embed data is proportional to the size of the cover image.

Additionally, we ran microbenchmarks on `gzip` to determine its computational requirements, as well as the compression ratios it achieves on typical HTML files. We fetched the `index.html` page from 20 popular Web sites that we selected from Nielsen Netratings [8]. `gzip` compressed these HTML files to as much as 16% of their original size; in the worse case, `gzip` compressed the HTML file to 74% of its original size. In all cases, `gzip` compression is not a bottleneck in content preparation. Table 1 shows the detailed results.

6.2 Server Bandwidth

To ensure plausible deniability for Infranet clients, Infranet servers always embed data in the content they serve. Because the server makes no distinction between Infranet clients and normal Web clients, an Infranet server incurs additional overhead in serving data to *all* clients.

Therefore, to determine the performance implications of running an Infranet server, we submit an identical sequence of requests to both an Infranet enabled server, and an ordinary Apache server. We measure the difference in performance between both servers when serving that trace.

We used two traces in the experiments. Both are traces from an Infranet browsing session generated by using an Infranet client to access a set of URLs and dumping the HTTP access log. One trace is an emulation of Infranet traffic using `gzip` compression of retrieved HTML data; the other trace reflects an Infranet session without compression.

Figure 3 shows the effects of running Infranet on server bandwidth. Running Infranet, even with `gzip` compression, incurs a significant performance penalty: 30% of all requests served on an Infranet server achieved 300 kB/s or less, and 95% of requests were transferred at 1 MB/s or less. In contrast, 50% of requests on a normal Apache server were delivered at 1 MB/s or greater. Nevertheless, for disk bound workloads, an Infranet server running with `gzip` compression performs comparably to a regular Apache server. Bandwidth achieved by Infranet never drops below 35% of that achieved by an Apache server running without Infranet.

An Infranet server that employs `gzip` compression of HTML data increases bandwidth over an Infranet server that does not employ compression compression. For some content, compression can increase server bandwidth by as much as a factor of 10. Nevertheless, even with compression, running Infranet halves server bandwidth in the common case.

6.3 End-to-end Performance

In this section, we conduct end-to-end performance tests by requesting popular Web sites both through Infranet and directly. We also evaluate the performance benefits of compressing HTML content using `gzip`. To conduct our experiments, we downloaded the top-level pages from 20 popular Web sites, most of which we selected from Nielsen Netratings' Top 25 for the week of November 25 [8]. For each site, we made 10 requests directly to the site and 10 requests through Infranet, tracking end-to-end download times for each.

Figure 4 shows the average end-to-end download times for these pages, plotted against their sizes. The graph shows this relationship for three different experiments—downloading content directly ("normal" Web browsing), through Infranet with no compression, and through Infranet with `gzip` compression of HTML files. The figure shows that, while compression helps download times, Infranet browsing is still significantly slower than normal Web browsing. The high degree of variability in the direct download times stems from the different network conditions that exist on the path

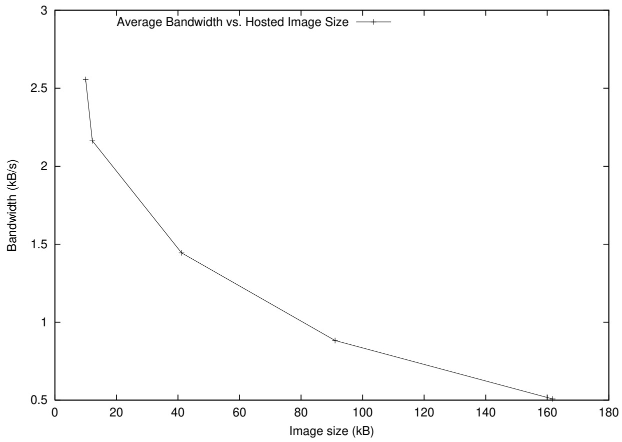


Figure 5. Change in average end-to-end bandwidth of covert content as an Infranet server's image sizes change. As an Infranet server's hosted images grow, the bandwidth provided for covert content drops off exponentially.

State	*No Gzipping*	*Gzipping*
Receive Dictionary	2.446433	0.603254

Table 2. End-to-end times (in seconds) for transmitting a dictionary update. `gzip` **compression significantly increases efficiency.**

to each Web site.

One important aspect is that `gzip` compression increases download times by roughly a factor of ten over browsing through Infranet with no compression. The main disadvantage of compressing all content is that compressed content cannot be streamed from the Infranet client to the user's browser as it is received.

To determine the correlation between average end-to-end bandwidth through Infranet and cover image sizes, we replaced every image on the Infranet server with with one image. We then varied the size of this image to determine how end-to-end download times vary for popular Web sites. Figure 5 shows the relationship between the average end-to-end bandwidth through Infranet as a function of the cover image size. Larger files can hold more hidden data. Conversely, large files take longer to process and transmit.

6.4 Performance breakdown by state

In this section we examine the time spent in each part of the Infranet protocol to determine performance bottlenecks. We consider both startup costs and steady state performance.

Table 3 shows the breakdown of how much time the server spends in each state of the FSM for a request in steady state. To compute these numbers, we use the same test as that in Section 6.3. When compression is not used, downloading content from the server is a significant bottleneck. However, when compression is used, other states, such as URL-keying, contribute a significant amount of time to total download time.

State	*No Gzipping*	*Gzipping*
URL-Keying	3.887108	3.815172
Retrieving Content	22.986341	2.648829
Send EOF	1.272909	1.197214
Total	28.146359	7.661215

Table 3. Average time (in seconds) spent in each state for steady state operation. Compressing content increases bandwidth considerably in the fetch state, because it drastically reduces the amount of data that must be embedded and sent to the client.

6.4.1 Key Exchange

When the client connects to the server for the first time, it transmits a seed to the server that will be used to generate the shared secret. Because this is a fixed cost that only needs to be done once per client-server pair, we decided to forgo the implementation of this feature in favor of analyzing characteristics of common operation.

6.4.2 Dictionary Update

After the key exchange, the server sends a dictionary update to the client, encoded with the shared secret. The duration of this process depends for the most part on the size of the dictionary. Table 2 shows the time required to send a dictionary update. We used a simple dictionary with approximately 100 entries. As expected, compression of the dictionary significantly reduces the time required for this state.

6.4.3 URL-keying

In our current implementation, the dictionary is static. Thus, the client achieves approximately 1 byte/request of upstream bandwidth because the server and client are not making effective use of the URL completion functionality. The client's upstream bandwidth could be significantly increased by a more intelligent URL completion strategy, as discussed in Section *3.1.2*.

If a client were to request only HTML pages during URL-keying and only embedded images during data retrieval, browsing would look suspicious. Therefore, a client must request embedded images for requested pages, even though these requests do not correspond to dictionary entries. This constraint results in a significant bandwidth reduction. Figure 6 shows that, in the common case, about 25% of requests made during URL-keying are for embedded images. 35% of all hidden requests result in more than half requests made in URL-keying for retrieving embedded images.

6.4.4 Retrieving content

Figure 7 shows the number of requests that a user must make to receive data of various sizes. Obviously, the number of requests required to retrieve content is proportional to the size of the content. Compression reduces the number of requests that a client must make by roughly a factor of 4.

As shown on Table 3 compression increases the downstream bandwidth from the server by as much as a factor of 10. The benefits of compression are due to two factors. First, sending more data obviously take more time. Second, data compression speeds up the transmission of hidden content because the server does not need to embed data into as many images, an operation that takes a

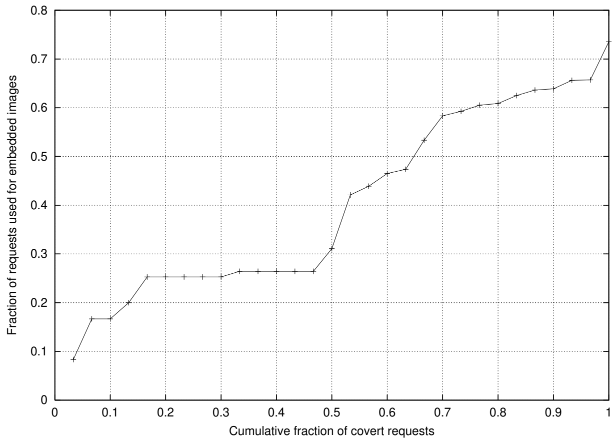


Figure 6. Fraction of requests used for embedded images when URL-keying a covert message. The fraction of requests varies greatly between 10% and 75%. In the common case, about 25% of requests made during URL-keying are for embedded images.

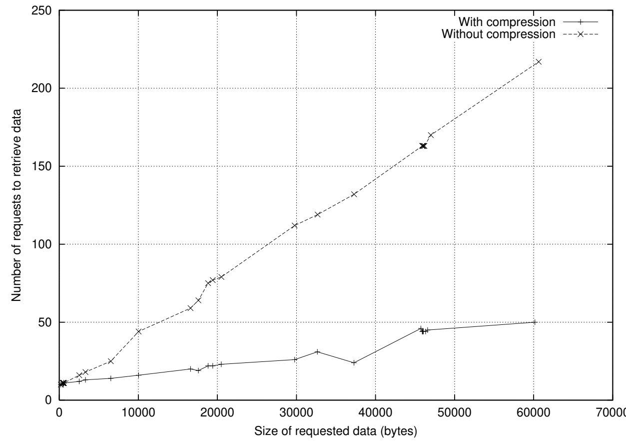


Figure 7. Number of requests required to retrieve data of various sizes. The number of requests grows linearly with data size. Using compression reduces the amount of requests roughly by a factor of four.

significant amount of time.

Figure 8 shows that, while compression helps download times considerably, it does not reduce the number of requests that the client must make as much as one might expect. When using compression, the number of requests that the client must make during the fetch state is only reduced by a factor of four, even though download times are smaller by a factor of 10. This is because there is a fixed cost for downloading content, independent of message size, because many requested links are not suitable for embedding content (e.g., they might be requests for HTML, etc.).

Table 3 also shows the significant cost of sending an end-of-file (EOF). The EOF marker is used to indicate that the server is finished sending embedded content to the client. We implemented this by creating a file that contains the delimiter, and embedding the character in an image using `outguess`. The server incurs considerable overhead for sending an EOF character in a separate image. This performance penalty becomes increasingly significant for small file transfers. A more appropriate solution would be to include a content header with every message.

7. Related work

Previous research has also looked at the problem of circumventing the surveillance of Internet traffic. Some basic schemes prevent the disclosure of client-identifying information [19]. These systems employ techniques such as firewalls and disabling cookies. However, they do not address the issue of concealing what content is requested and received,nor do they provide any means of defeating censorship.

Many systems provide anonymity by requiring the client to make requests through an anonymizing proxy [1, 3, 11, 12]. Safeweb is an example of a proxy-based approach to achieving anonymity. Safeweb runs a proxy that allows a client to protect its identity from the site it is browsing. The proxy also defeats firewall filtering by having the Safeweb request the restricted page [11]. The proxy retrieves the content and sends it back to the user, encrypted over an SSL channel. “Triangle Boy” is Safeweb’s response to various organizations’ attempts to restrict access to the Safeweb proxy. Triangle Boy is a mechanism by which HTTP requests can be made to any PC running the Triangle Boy software [17]. Requests are are returned to the user via an SSL connection from the Safeweb proxy, which spoofs the source IP address to make it appear that the content originated from that PC. The Safeweb approach has several limitations. First, the user’s browser history and cache reveal a user’s browsing habits. Second, the use of an SSL channel engenders suspicion. Finally, Triangle Boy’s reliance on IP source address spoofing is not likely to work, given that many routers perform ingress filtering.

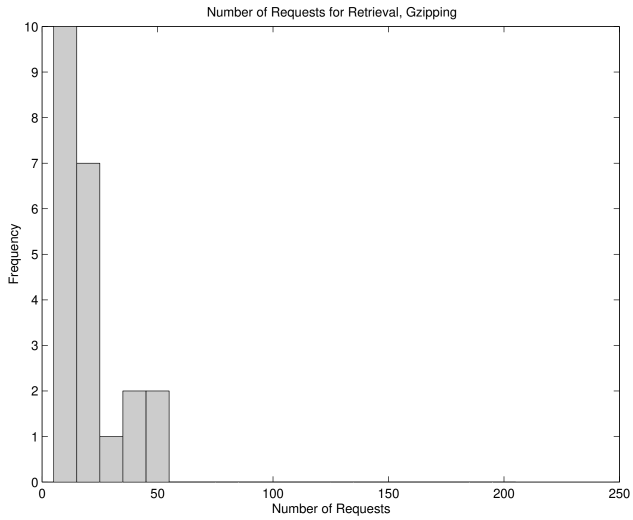
Peer-to-peer efforts comprise another approach to protecting Internet identity. One such system, Crowds [10], joins users into a large, geographically diverse group whose members cooperate in issuing requests. This approach makes it difficult to associate requests with the originating user. The onion routing scheme also works to separate requests from those who made them [14].

Peekabooty is a forthcoming distributed file sharing application that hides both the shared data and the participants sharing that data [16]. Similar to Safeweb, Peekabooty also encrypts communication and hence has the same disadvantage of raising suspicion. Additionally, Peekabooty relies on at least one user being able to access the restricted content.

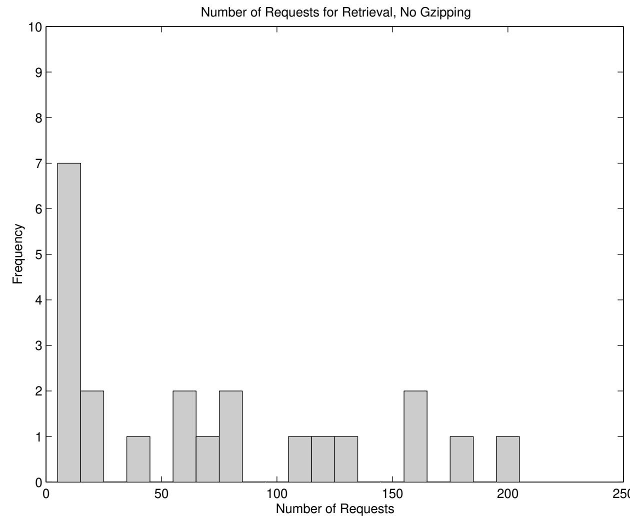
8. Conclusion

Infranet enables users to circumvent Web censorship and surveillance by establishing covert channels with accessible Web servers. Clients compose secret messages using sequences of innocuous requests, and servers respond by embedding data in the openly requested content. The resulting traffic resembles the traffic generated by normal browsing. Hence, Infranet provides both access to sensitive content and plausible deniability to users.

In our prototype implementation, an Infranet client downloads at 3 to 5 kB/s. Bandwidth is largely dependent on the size of images used to embed covert content. Furthermore, our evaluation



(a) With compression.



(b) Without compression

Figure 8. Number of requests required to download popular web pages. Because not all requests are amenable to embedding hidden content, the user must often make many requests, even when the requested data is compressed.

indicates that the performance of an Infranet-enabled Web server is comparable to an ordinary Apache server satisfying all requests from disk.

We expect several future enhancements to strengthen the system. Maintaining a relevant mapping from URLs to message fragments will both increase upstream bandwidth and improve network traffic patterns. In addition, a more efficient steganographic utility will increase downstream bandwidth.

Acknowledgments

We are very grateful to Hari Balakrishnan and David Karger for this project idea and many helpful design suggestions and discussions. We also thank David Karger for coining the term "Infranet." Thank you also to Dave Andersen for several helpful discussions. We also thank Robert Morris and Frank Dabek for their comments.

9. REFERENCES

[1] Anonymizer. Anonymous Web Surfing and Anonymizer Secure Tunneling. `http://www.anonymizer.com`.

[2] Crypto++. `http://www.eskimo.com/~weidai/cryptlib.html`.

[3] E. Gabber, P. Gibbons, Y.Matias, and A. Mayer. How to make personalized web browsing simple, secure, and anonymous. In *Financial Cryptography 97*, 1997. `http://www.bell-labs.com/project/lpwa`.

[4] List Guernsey. Court says france can't censor yahoo site, November 9, 2001. `http://www.nytimes.com/2001/11/09/technology/09YAHO.html`.

[5] The gzip home page. `http://www.gzip.org`.

[6] B.W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[7] Paul Meller. Europe moving toward ban on internet hate speech, November 10, 2001. `http://www.nytimes.com/2001/11/10/technology/10CYBE.html`.

[8] Nielsen netratings' top 25. `http://pm.netratings.com/nnpm/owa/NRpublicreports.toppropertiesweekly`, November 25 2001.

[9] Outguess. `http://www.outguess.org`.

[10] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security (TISSEC)*, 1, 1998. `http://www.research.att.com/projects/crowds/`.

[11] SafeWeb. `http://www.safeweb.com`.

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

[13] Lincoln Stein et al. *Writing Apache Modules with Perl and C: The Apache API and mod_perl*. O'Reilly and Associates, March 1999.

[14] Paul F. Syverson, Michael G. Reed, and David M. Goldschlag. Onion routing access configurations. In *DARPA Information Survivability Conference and Exposition*, pages 34–40, 2000. `http://www.onion-router.net`.

[15] R. Thayer and K. Kaukonen. A stream cipher encryption algorithm, 1997.

[16] The Cult of the Dead Cow (cDc). Peekabooty. `http://www.vnunet.com/News/1121286`.

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

[18] H. C. Williams. A modification of the RSA public-key encryption procedure. *IEEE Transactions on Information Theory*, IT-26(6):726–729, 1980.

[19] Zero-Knowledge Systems. Freedom Privacy and Security Tools. `http://www.zeroknowledge.com/`.