SSL Splitting and Barnraising: Cooperative Caching with Authenticity Guarantees

by

Christopher T. Lesniewski-Laas

S.B., Massachusetts Institute of Technology (2001)

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

February 2003

© Christopher T. Lesniewski-Laas, MMIII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly
paper and electronic copies of this thesis and to grant others the right to do so.

Author .....
Department of Electrical Engineering and Computer Science
February 4, 2003

Certified by .....
M. Frans Kaashoek
Professor of Computer Science and Engineering
Thesis Supervisor

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

SSL Splitting and Barnraising: Cooperative Caching with Authenticity Guarantees

by

Christopher T. Lesniewski-Laas

Submitted to the Department of Electrical Engineering and Computer Science
on February 4, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

SSL splitting is a cryptographic technique to guarantee that public data served by caching Web proxies is endorsed by the originating server. When a client makes a request, the trusted server generates a stream of authentication records and sends them to the untrusted proxy, which combines them with a stream of data records retrieved from its local cache. The combined stream is relayed to the client, a standard Web browser, which verifies the data's integrity. Since the combined stream simulates a normal Secure Sockets Layer (SSL) [7] connection, SSL splitting works with unmodified browsers; however, since it does not provide confidentiality, it is appropriate for applications that require only authentication. The server must be linked to a patched version of the industry-standard OpenSSL library; no other server modifications are necessary. In experiments replaying two-hour `access.log` traces taken from LCS Web sites over a DSL link, SSL splitting reduces bandwidth consumption of the server by between 25% and 90% depending on the warmth of the cache and the redundancy of the trace. Uncached requests forwarded through the proxy exhibit latencies within approximately 5% of those of an unmodified SSL server.

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Computer Science and Engineering

Acknowledgements

This work was supported by MIT Project Oxygen.

My heartfelt gratitude goes to:

Frans, for his unflagging encouragement, wisdom, and generosity;

Jacob, for sound advice and a ready ear;

Jessica, Rachel, Roger, Gerald, Grace, Amittai, and Danielle, for nobly sacrificing their network so that this thesis could live;

The `www.lcs.mit.edu` webmasters, and especially Noah Meyerhans, for kindly accommodating my requests for data;

The curmudgeons of SIPB, for long incubating my pragmatic mindset and for sharing their prodigous expertise;

My parents, for always believing in me;

Malima, for her love and support.

Contents

List of Figures

Chapter 1

Introduction

Caching Web proxies are a proven technique for reducing the load on centralized servers. In existing systems, however, each proxy must be trusted by both the client and the server to return the correct results to the client's queries. SSL splitting, a simple approach to this problem, provides an end-to-end integrity guarantee to the client and server, while allowing the server's data stream to be served from a proxy's cache. Unlike previous cryptographic approaches to authenticated content distribution, SSL splitting uses the Transport Layer Security support built into standard Web browsers. Since it does not require any special-purpose client software to use, SSL splitting is easier to deploy than others.

An SSL splitting proxy works by downloading a stream of SSL record authenticators from a trusted central server, and merging them with a stream of message payloads retrieved from the proxy's cache. The merged data stream, sent to the client, is indistinguishable from a normal SSL conversation between the client and the server. This simple approach cleanly separates the roles of the server and the proxy: the server, as "author", originates and signs the correct data, and the proxy, as "distribution channel", serves the data to clients.

Our primary application for SSL splitting is *Barnraising*, a cooperative Web cache. We anticipate that secure cooperative content delivery will be useful to bandwidth-hungry Web sites with limited central resources, such as software distribution archives and media artists' home pages. Currently, such sites must be mirrored by people known and trusted by the authors, since a malicious mirror can sabotage the content; Barnraising would allow such sites to harness the resources of arbitrary hosts on the Internet. Barnraising could also be

used by *ad hoc* cooperatives of small, independent Web sites, to distribute the impact of localized load spikes.

Previous content delivery systems that guarantee the integrity of the data served by replicas require changes to the client software (e.g., to support SFSRO [8]), or use application-specific solutions (e.g., RPM with PGP signatures [22]). The former have not seen wide application due to lack of any existing client base. The latter are problematic due to PKI bootstrapping issues and due to the large amount of manual intervention required for their proper use. By taking advantage of widely-deployed browser support for the SSL protocol, SSL splitting provides integrity protection without placing a heavy burden on users of the system.

The contributions of this thesis are (1) the design of the SSL splitting technique, including the simple protocol between server and proxy to support SSL splitting (see Chapter 2); (2) an implementation of SSL splitting based on the freely available OpenSSL library (see Chapter 3); (3) a new, grassroots content-distribution system, *Barnraising*, which applies SSL splitting (see Chapter 4) to distribute bandwidth load; and (4) experiments that show that SSL splitting has CPU costs similar to SSL, but saves server bandwidth, and improves download times for large files (see Chapter 5). We also relate SSL splitting to previous work (see Chapter 7) and draw our conclusions in Chapter 8.

Chapter 2

Design of SSL splitting

The key idea behind SSL splitting is that a stream of SSL records is separable into a data component and an authenticator component. As long as the record stream presented to the client has the correct format, the two components can arrive at the proxy by different means. In particular, a proxy can cache data components, avoiding the need for the server to send the data in full for every client.

While SSL splitting does not require changes to the client software, it does require a specialized proxy and modifications to the server software. The modified server and proxy communicate using a new protocol that encapsulates the regular SSL protocol message types and adds two message types of its own.

2.1 Goals

The main goal of SSL splitting is to guarantee that public data served by caching Web proxies is endorsed by the originating server. Anybody with an inexpensive DSL line should be able to author content and distribute it from his own Web server. To allow this limited connection to support a higher throughput, authors can leverage the resources of well-connected volunteers acting as mirrors of the site's content. However, since the authors may not fully trust the volunteers, SSL splitting must provide an end-to-end authenticity and freshness guarantee: the content accepted by a client must be the latest version of the content published by the author.

The second goal of SSL splitting it to provide the data integrity with minimal changes to the existing infrastructure. More specifically, our goal is a solution that does not require

any client-side changes and minimal changes to a server. To satisfy this goal, SSL splitting exploits the existing support for SSL.

While SSL splitting ensures the integrity of content sent via proxies, confidentiality is not a goal. SSL splitting's intended use is to distribute public, popular data from bandwidth-limited servers. Most of the content on the Web is not secret, and moreover, most pages that require secrecy are dynamically generated and hence not cacheable. Lack of confidentiality is also an inevitable consequence of exploiting SSL, since SSL's encryption provides ciphertext indistinguishability: any caching proxy must be able to tell when two clients have downloaded the same file.

SSL splitting does not provide all the benefits of traditional, insecure mirroring. While it improves the bandwidth utilization of the central site, it incurs a CPU load similar to a normal SSL server. In addition, it does not improve the redundancy of the site, since the central server must be available in order to authenticate data. Redundant central servers must be employed to ensure continued service in the face of server failure or network partition.

2.2 SSL overview

The Secure Sockets Layer protocol provides end-to-end mutual authentication and confidentiality at the transport layer of stream-based protocols [7]. A typical SSL connection begins with a handshake phase, in which the server authenticates itself to the client and shared keys are generated for the connection's symmetric ciphers. The symmetric keys generated for authentication are distinct from those generated for confidentiality, and the keys generated for the server-to-client data stream are distinct from those generated from the client-to-server stream.

After completing the handshake, the server and client exchange data asynchronously in both directions along the connection. The data is split into records of 2^{14} bytes or less. For each record, the sender computes a Message Authentication Code (MAC) using the symmetric authentication keys; this enables the receiver to detect any modification of the data in transit. SSL can provide confidentiality as well as integrity: records may be encrypted using the shared symmetric encryption keys. Although SSL generates the keys for both directions from the same "master secret" during the handshake phase, the

two directions are subsequently independent: the client or server's outgoing cipher state depends only on the previous records transmitted by that party.

2.3 Interposing a proxy

Figure 2-1 shows the flow of data from server to proxy to client when using SSL splitting; it represents the server-to-client stream of the SSL connection. Initially, a Web browser contacts a proxy using HTTP over SSL/TLS (HTTPS [16]). The server may have redirected the client to the proxy via any mechanism of its choice, or the proxy may have already been on the path between the browser and the server.

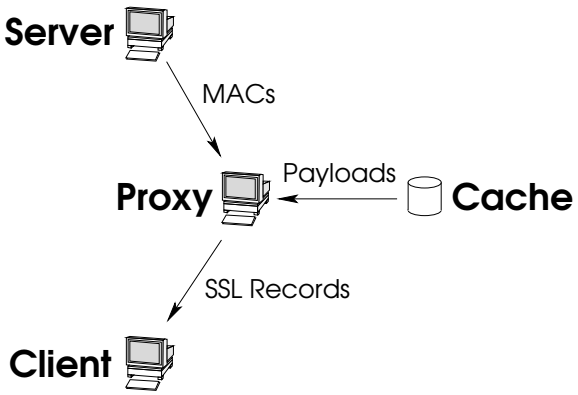

Figure 2-1: Data flow in SSL splitting

The proxy relays the client's connection setup messages to the server, which in turn authenticates itself to the client via the proxy. Once the SSL connection is set up, the server starts sending application data: for each record, it sends the message authentication code (MAC) along with a short unique identifier for the payload. Using the identifier, the proxy looks up the payload in its local cache, splices this payload into the record in place of the identifier, and relays this reconstructed record to the client. The client verifies the integrity of the received record stream, which is indistinguishable from the stream that would have been sent by a normal SSL server.

Since SSL is resistant to man-in-the-middle attacks, and the proxy is merely a man-in-the-middle with respect to the SSL handshake, the SSL authentication keys are secret from the proxy. Only the server and the client know these keys, which enable them to generate and verify the authentication codes protecting the connection's end-to-end integrity and

freshness.

2.4 Proxy-server protocol extensions

When a client initiates an HTTPS connection to an SSL-splitting proxy, the proxy immediately connects to the server using the specialized proxy-server protocol. This protocol defines three message types. The first type of message, *verbatim*, is a regular SSL record, passed transparently through the proxy from the client to the server, or vice versa. The second type of message, *stub*, is a compact representation of an SSL record: it contains a MAC authenticator and a short unique identifier for the payload. The third type of message, *key-expose*, communicates an encryption key from the server to the proxy.

When the proxy receives SSL records from the client, it forwards them directly to the server using the *verbatim* message. The server, however, may choose to compress the data it sends by using the *stub* message format. When the proxy receives such a message from the server, it looks up the data block identified by the message in its local cache. It then reconstructs a normal SSL record by splicing the MAC authenticator from the stub record together with the payload from the cache, and forwards the resulting valid SSL record to the client.

2.5 Dropping the encryption layer

A proxy can properly forward *stub* messages only if it is able to encode the resulting normal SSL records. If the client and server use SSL with its end-to-end encryption layer enabled, however, the proxy cannot send validly encrypted messages. Inherently, end-to-end encryption always foils caching, because a proxy which forwards the same block to multiple clients knows that all the clients have received the same data; this contradicts indistinguishability. Therefore, to achieve bandwidth compression, confidentiality—with respect to the proxy—must be abandoned.

The correct way to eliminate SSL's encryption layer is to negotiate, during the handshake phase, an authentication-only cipher suite such as `SSL_RSA_WITH_NULL_SHA`; this is usually done with a Web server configuration setting. When such a cipher suite is in use, no confidentiality is provided for SSL records sent in either direction; only data authentication is provided.

Unfortunately, this straightforward approach does not achieve full compatibility with the existing installed client base, because current versions of many popular Web browsers, such as Netscape and Internet Explorer, ship with authentication-only cipher suites disabled. The SSL splitting protocol provides a work-around for this problem, which can be enabled by the server administrator. If the option is enabled and a client does not offer an authentication-only cipher suite, the server simply negotiates a normal cipher suite with the client, and then intentionally exposes the server-to-client encryption key and initialization vector (IV) to the proxy using the *key-expose* message.

The encryption key, IV, and MAC key are generated as independent pseudo-random functions (PRF) of the master secret. Because SSL uses different PRFs for each key, revealing the cipher key and IV to the proxy does not endanger the MAC key or any of the client-to-server keys [17, p. 165]. Hence, this work-around preserves the data-authentication property.

When the *key-expose* message is turned on and an encrypted cipher suite is negotiated, the client-to-server encryption keys are withheld from the proxy; thus, it would be possible to develop an application in which the information sent by a client was encrypted, while the content returned by the server was unencrypted and cacheable by the proxy. However, unless the application is carefully designed, there is a danger of leaking sensitive data in the server's output, and so we do not recommend the use of SSL splitting in this mode without very careful consideration of the risks.

If necessary, an application can restrict the set of hosts with access to the cleartext to the client, the server, and the proxy, by encrypting the proxy-server connection. For example, one could tunnel the proxy-server connection through a (normal) SSL connection. Of course, this feature would be useful only in applications where the proxy can be trusted not to leak the cleartext, intentionally or not; as above, this is a mode of operation that we do not recommend.

2.6 Server-proxy signaling

SSL splitting does not mandate any particular cache coherency mechanism, but it does affect the factors that make one mechanism better than another. In particular, caching with SSL splitting is at the SSL record level rather than at the file level. In theory, a single

file could be split up into records in many different ways, and this would be a problem for the caching mechanism; however, in practice, a particular SSL implemention will always split up a given data stream in the same way.

Deciding which records to encode as *verbatim* records and which to encode as *stub* records can be done in two ways. The server can remember which records are cached on each proxy, and consult an internal table when deciding whether to send a record as a stub. However, this has several disadvantages. It places a heavy burden on the server, and does not scale well to large numbers of proxies. It does not give the proxies any latitude in deciding which records to cache and which to drop, and proxies must notify the server of any changes in their cached set. Finally, this method does not give a clear way for the proxies to initialize their cache.

Our design for SSL splitting uses a simpler and more robust method. The server does not maintain any state with respect to the proxies; it encodes records as *verbatim* or *stub* without regard to the proxy. If the proxy receives a *stub* that is not in its local cache, it triggers a cache miss handler, which uses a simple, HTTP-like protocol to download the body of the record from the server. Thus, the proxies are self-managing and may define their own cache replacement policies. This design requires the server to maintain a local cache of recently-sent records, so that it will be able to serve cache miss requests from proxies. Although a mechanism similar to TCP acknowledgements could be used to limit the size of the server's record cache, a simple approach that suffices for most applications is to pin records in the cache until the associated connection is terminated.

The cache-miss design permits the server to use an arbitrary policy to decide which records to encode as *stub* and hence to make available for caching; the most efficient policy depends on the application. For HTTPS requests, an effective policy is to cache all application-data records that do not contain HTTP headers, since the headers are dynamic and hence not cachable. Most of the records in the SSL handshake contain dynamic elements and hence are not cachable; however, the server's certificates are cachable. Caching the server certificates has an effect similar to that of the "fast-track" optimization described in [19], and results in an improvement in SSL handshake performance. This caching is especially beneficial for requests of small files, where the latency is dominated by the SSL connection time.

Since the *stub* identifier is unique and is not reused, there is no need for a mechanism

to invalidate data in the cache. When the file referenced by an URL changes, the server sends a different stream of identifiers to the proxy, which does not know anything about the URL at all. In contrast, normal caching Web proxies [21] rely on invalidation timeouts for a weak form of consistency.

Chapter 3

Implementation

Our implementation of SSL splitting consists of a self-contained proxy module and a patched version of the OpenSSL library, which supports SSL version 3 [7] and Transport Layer Security (TLS) [4].

3.1 Server: modified OpenSSL

We chose to patch OpenSSL, rather than developing our own protocol implementation, to simplify deployment. Any server that uses OpenSSL, such as the popular Web server Apache, works seamlessly with the SSL splitting protocol. In addition, the generic SSLizing proxy STunnel, linked with our version of OpenSSL, works as an SSL splitting server: using this, one can set up SSL splitting for servers that don't natively understand SSL. This allows SSL splitting to be layered as an additional access method on top of an existing network resource.

Our modified version of OpenSSL traps the record-encoding routine `do_ssl_write` and analyzes outgoing SSL records to identify those that would benefit from caching. Our current implementation tags non-header application-data records and server certificate records, as described in Section 2.6.

Records that are tagged for caching are hashed to produce a short `digest` payload ID, and the bodies of these records are *published* to make them available to proxies that do not have them cached. While publishing is a modular operation that may take many application-specific forms, our implementation simply writes these payloads into a cache directory on the server; a separate daemon process serves this directory to proxies.

Records that are not tagged for caching use the `literal` payload "ID" encoding. Whether or not the record is tagged for caching, the library encodes the record as a *stub* message; this is purely because it simplifies the code.

Because the server may have to ship its encryption key and IV to the proxy, the modified OpenSSL library contains additional states in the connection state machine to mediate the sending of the *key-expose* message. The server sends this message immediately after any `change_cipher_spec` record, since this is when the connection adopts a new set of keys.

3.2 Proxy

The proxy is simple: it forks off two processes to forward every accepted connection. It is primarily written in OO Perl5, with the performance-critical block cipher and CBC mode implementation in C. The proxy includes a pluggable cache hierarchy: when a cache lookup for a payload ID fails, it can poll outside sources, such as other proxies, for the missing data. If all else fails, the server itself serves as an authority of last resort. Only if none of these have the payload will the proxy fail the connection.

The proxy could also replace *verbatim* messages from the client to the server with *stub* messages. Because the client's data consists primarily of HTTP GET requests, however, which are already short and are not typically repeated, our current proxy doesn't do so. In the future, though, we may explore a proxy that compresses HTTP headers using *stub* messages.

3.3 Message formats

Figures 3-1 and 3-2 show the format of *verbatim* and *stub* message in the same notation as the SSL specification. The main difference between *verbatim* and *stub* is that in *stub* the payload is split into a compact encoding of the data and a MAC authenticator for the data. Also, a stub record is never encrypted, since the proxy would have to decrypt it anyway in order to manipulate its contents.

We have defined two types of encoding for the payload. The *literal* encoding is the identity function; this encoding is useful for software design reasons, but is functionally equivalent to a *verbatim* SSL record.

```
enum { ccs(0x14), alert(0x15), handshake(0x16), data(0x17) } ContentType;
struct {
 ContentType     content_type;           one byte long
 uint8           ssl_version[2];
 opaque          encrypted_data_and_mac<0..2^14+2048>;
                                         includes implicit 2-byte length field
} VerbatimMessage;
```

Figure 3-1: Format of a verbatim SSL record

```
enum { s_ccs(0x94), s_alert(0x95), s_handshake(0x96), s_data(0x97) } StubContentType;
enum { literal(1), digest(2), (2^16-1) } IDEncoding;
struct {
 StubContentType content_type;           = verbatim.content_type | 0x80
 uint8           ssl_version[2];
 uint16          length;                 = 6 + length(id) + length(mac)
 IDEncoding      encoding;
 opaque          id<0..2^14+2048>;
 opaque          mac<0..2^14+2048>;
} StubMessage;
```

Figure 3-2: Format of a stub message

The *digest* encoding is a SHA-1 [5] digest of the payload contents. This encoding provides effectively a unique identifier that depends only on the payload. This choice is convenient for the server, and allows the proxy to cache payloads from multiple independent servers in a single cache without concern about namespace collisions.

There are many alternative ID encodings possible with the given *stub* message format; for example, a simple serial number would suffice. The serial number, however, has only weak advantages over a message digest. A serial number is guaranteed to be unique, unlike a digest. On the other hand, generating serial numbers requires servers to maintain additional state, and places an onus upon proxies to separate the caches corresponding to multiple servers; both of these result in greater complexity than the *digest* encoding.

Another alternative is to use as the encoding a compressed representation of the payload. While this choice would result in significant savings for text-intensive sites, it would not benefit image, sound, or video files at all, since most media formats are already highly compressed. For this reason, we have not implemented this feature.

The *key-expose* message is used to transmit the server encryption key and IV to the

proxy (see Figure 3-3) if this feature is enabled. In our implementation, *stub* records are sent in the clear to the proxy, which encrypts them before sending them to the client.

```
enum { key_expose(0x58) } KeyExposeContentType;
struct {
 KeyExposeContentType content_type;
 uint8                ssl_version[2];
 uint16               length;            = 4 + length(key) + length(iv)
 opaque               key<0..2^14+2048>;
 opaque               iv<0..2^14+2048>;
} KeyExposeMessage;
```

Figure 3-3: Format of key-expose message

Chapter 4

Cooperative Web caching using SSL splitting

Using SSL splitting, we have developed *Barnraising*, a cooperative Web caching system consisting of a dynamic set of "volunteer" hosts. The purpose of this system is to improve the throughput of bandwidth-limited Web servers by harnessing the resources of geographically diverse proxies, without trusting those proxies to ensure that the correct data is served.

4.1 Joining and leaving the proxy set

Volunteers can join or leave the proxy set of a site by contacting a *broker* server, which maintains the volunteer database and handles redirecting client requests to volunteers.

Volunteer hosts do not locally store any configuration information, such as the SSL splitting server's address and port number. These parameters are supplied by the broker in response to the volunteer's `join` request, which simply specifies an identifying URI of the form `barnraising://broker.domain.org/some/site/name`.

This decision enables a single broker to serve any number of Barnraising-enabled Web sites, and permits users to volunteer for a particular site given only a short URI for that site. Since configuration parameters are under the control of the broker, they can be changed without manually reconfiguring all proxies, allowing sites to upgrade transparently.

The broker represents a bottleneck for the system, and it could be swamped by a large number of simultaneous join or leave requests. However, since the join/leave protocol is lightweight, this is unlikely to be a performance issue under normal operating conditions.

If the load incurred by requests is high compared to the actual SSL splitting traffic, the broker can simply rate-limit them until the proxy set stabilizes at a smaller size.

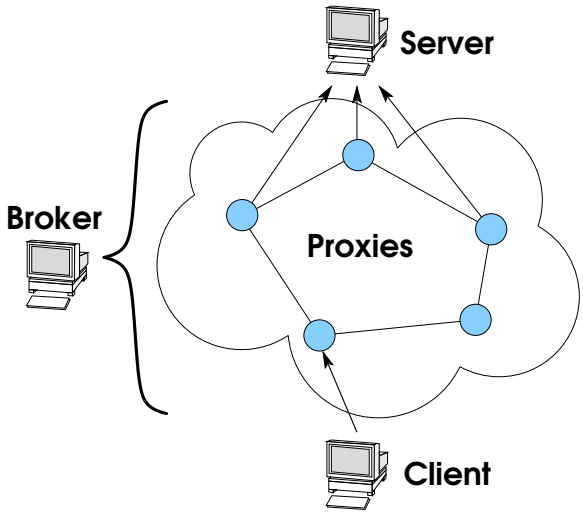

Figure 4-1: Proxy set for a site using Barnraising.

4.2 Redirection

Barnraising currently employs the DNS redirection method [11], but could be modified to support other techniques, such as URL rewriting [10, 1]. Barnraising's broker controls a `mysql` database, from which a `mydns` server processes DNS requests.

Consider a client resolving an URL `https://www.domain.org/foo/`. If `www.domain.org` is using Barnraising, the DNS server for `domain.org` is controlled by the broker, which will resolve the name to the IP address of a volunteer proxy.

We chose redirection using DNS because it maintains HTTPS reference integrity — that is, it guarantees that hyperlinks in HTML Web pages dereference to the intended destination pages. HTTPS compares the hostname specified in the `https` URL with the certificate presented by the server. Therefore, when a client contacts a proxy via a DNS name, the certificate presented to the client, by the server, via the proxy, must match the domain name. When DNS redirection is used, the domain name will be of the form `www.domain.org`, and will match the domain name in the certificate.

4.3 Distributing the Web cache

The client will initiate an HTTPS connection to the proxy, which will forward that request, using SSL splitting, to the server. Since frequently-accessed data will be served out of the proxy's cache, the central server's bandwidth usage will be essentially limited to the SSL handshake, MAC stream, and payload IDs.

To increase the set of cached payloads available to a proxy, while decreasing the local storage requirements, proxies share the cache among themselves. Volunteer nodes join a wide-area Distributed Hash Table (DHT) [3] comprising all of the volunteers for a given Web site. When lookups in the local cache fail, nodes attempt to find another volunteer with the desired data item by looking for the data in the DHT. If that fails too, the proxy contacts the central server.

Blocks in the DHT are named by the same cryptographic hash used for *stub* IDs. This decision allows correctly-operating volunteers to detect and discard any invalid blocks that a malicious volunteer might have inserted in the DHT.

4.4 Deploying Barnraising

Barnraising is designed to be initially deployed as a transparent layer over an existing Web site, and incrementally brought into the core software. Using STunnel linked with our patched OpenSSL library, an SSL splitting server which proxies an existing HTTP server can be set up on the same or a different host; this allows Barnraising to be tested without disrupting existing services. If the administrator later decide to move the SSL splitting server into the core, he can use Apache linked with SSL-splitting OpenSSL.

The broker requires a working installation of `mysql` and `mydns`; since it has more dependencies than the server, administrators may prefer to use an existing third-party broker while testing Barnraising. This brings the third party into the central trust domain of the server; much like a traditional mirror, it is a role which can only be filled by a reputable entity. However, unlike a major mirror, the network load incurred by a central broker does not include client downloads or periodic image refreshes, and so it carries less of the disincentives of running an archive mirror.

The utility of Barnraising will be limited by the volunteer proxies which join it. Therefore, the proxy has been designed to be simple to install and use, requiring only single URL

to configure. Since volunteers will be able to download from each other, they will have better performance than regular clients, giving users an incentive to install the software. In the long term, we hope to incorporate more efficient authentication schemes, such as SFSRO [8], into the volunteer code, using the installed base of SSL splitting software to bootstrap the more technically sophisticated systems.

Chapter 5

Evaluation

This section presents trace-based experiments to test the effectiveness and practicality of SSL splitting. The results of these experiments demonstrate that SSL splitting decreases the bandwidth load on the server, and that the performance with respect to uncached files is similar to vanilla SSL.

5.1 General experimental setup

For the experiments we used the Apache web server (version 1.3.23), linked with mod_ssl (version 2.8.7), and OpenSSL (version 0.9.6), running under Linux 2.4.18 on a 500 MHz AMD K6. This server's network connection is a residential ADSL line with a maximum upstream bandwidth of 160 kbps. The client was a custom asynchronous HTTP/HTTPS load generator written in C using OpenSSL, running under FreeBSD 4.5 on a 1.2 GHz Athlon. The proxy, when used, ran under FreeBSD 4.5 on a 700 MHz Pentium III. Both the client and the proxy were on a 100 Mbps LAN, with a 100 Mbps uplink.

5.1.1 Web trace files

The Web traces used in these experiments were derived from several-month `access.log` files of two departmental Web servers, `www.lcs.mit.edu` and `amsterdam.lcs.mit.edu`. To convert the access logs into replayable files, all non-GET, non-status-200 queries were filtered out, URLs were canonicalized, and every $(URL, size)$ pair was encoded as a unique URL. The server tree was then populated with files of random bytes.

Analyzing randomly-chosen chunks of various lengths from `www.lcs` showed that most

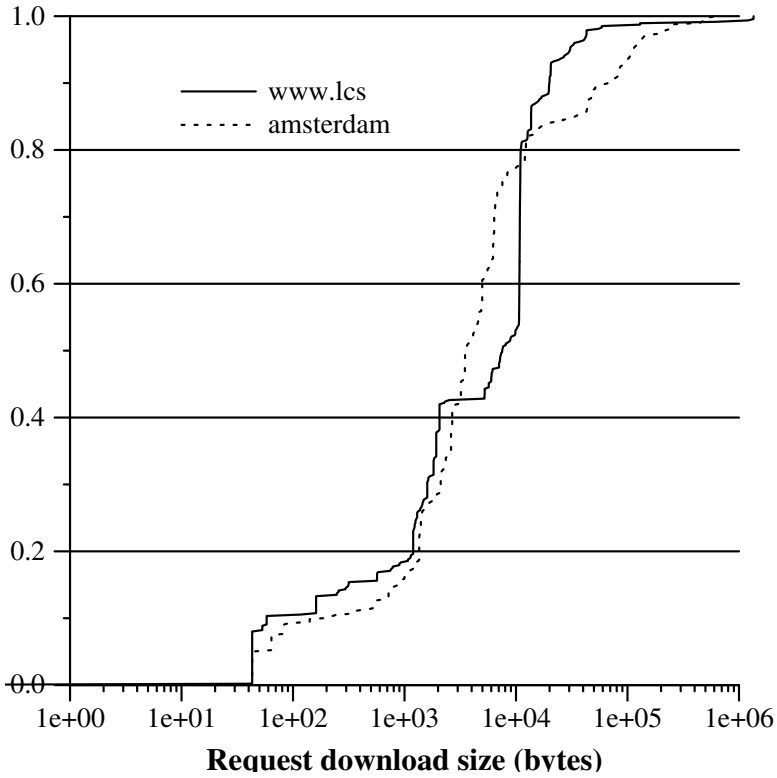

Figure 5-1: CDF of request sizes in `www.lcs` and `amsterdam` traces.

repetition is long-term: a day's trace (typically about 100 MB of data transfer) has a repetition factor which varies between 1.5 and 3, while a month is compressible by about a factor of 4, and the whole seven-month trace is compressible by a factor of 8. This suggests that having proxies keep blocks around for a long time will pay off, and supports a design in which proxies are organized into a DHT, since this allows them to keep more unique blocks for a longer time.

To keep experiment times managable over a DSL line, typical 10MB daytime chunks were selected from each trace. Figure 5-1 shows the distribution of request sizes in each trace chunk. The `www.lcs` chunk has an internal redundancy factor of 2.17, and the `amsterdam` chunk has an internal redundancy factor of 1.35.

5.2 Bandwidth savings

Ideally, SSL splitting's bandwidth utilization should be close to the inherent redundancy in the input trace. To test this, we played back the two 10MB trace chunks to a standard

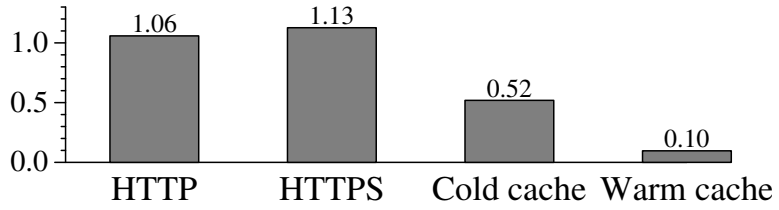

Figure 5-2: Bandwidth usage: `www.lcs` trace.

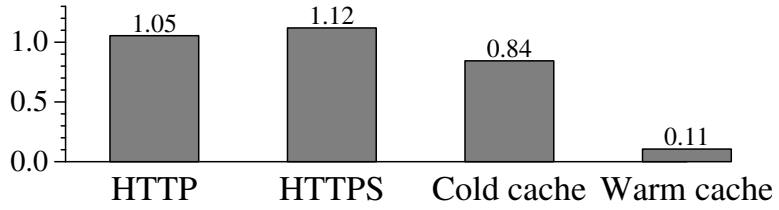

Figure 5-3: Bandwidth usage: `amsterdam` trace.

HTTP server, a standard HTTPS server, an SSL splitting proxy with a cold cache, and an SSL splitting proxy with a warm cache; in each case, we measured the total number of bytes sent on the server's network interface. The resulting bandwidth usage ratios (measured in bytes of bandwidth used per bytes of file throughput) are shown in Figures 5-2 and 5-3. As expected, SSL splitting with a cold cache achieves a compression ratio of 2.04 on `www.lcs` and 1.25 on `amsterdam`, with respect to HTTP; the compression is about 5% more with respect to HTTPS, very close to the ideal ratio. If the cache is warmed before running the trace, the compression ratio is approximately a factor of 11 for `www.lcs` and 10 for `amsterdam`. Analysis of the portions of the trace file preceding the `www.lcs` chunk indicate that if the previous two weeks had been cached by the proxy, the cache would have been approximately 90% warm. The `amsterdam` chunk is too close to the beginning of the trace to perform this analysis.

Since SSL splitting caches files at the record level, and a record takes a fixed cost to transmit, we would expect its ability to compress files would depend on the file size. A series of short microbenchmarks consisting of a single file download confirms this: SSL splitting is far more effective at caching large files. Figure 5-4 shows the bandwidth efficiency, calculated as the ratio of file throughput to bandwidth consumption, of HTTP, HTTPS, uncached SSL splitting, and cached SSL splitting. (The dotted line at 1 represents the

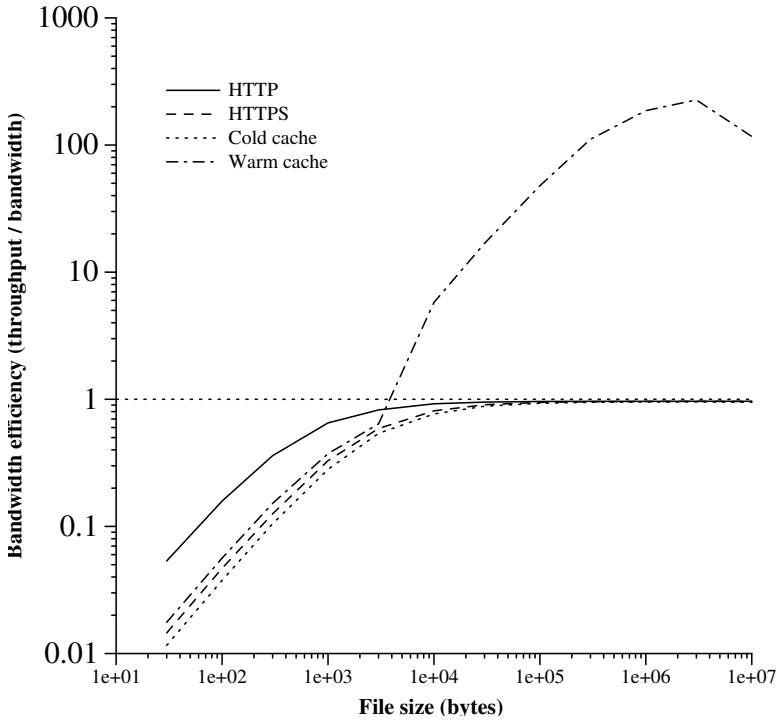

Figure 5-4: Ratio of throughput achieved to bandwidth used, when retrieving a single file from the server.

theoretical performance of a non-caching protocol with zero overhead.) Figure 5-5 shows this data as the "savings factor", the ratio of bandwidth consumed by SSL splitting (with a warm cache) to that consumed by HTTP or HTTPS. For 100-byte files, plain HTTP has about one-third the cost of SSL splitting, since the bandwidth cost for small files is dominated by connection setup; on the other hand, for one-megabyte files, SSL splitting has a bandwidth savings of 99.5% over HTTP.

There are two artifacts evident in these graphs. They show a sharp curve upward at the 3000 byte point; the reason for this artifact is that Apache sends all of the HTTP headers and file data in a single SSL record for files smaller than roughly 4000 bytes, but sends the HTTP headers in a separate record from the file data for larger files. There is also a sharp performance drop between 3 megabytes and 10 megabytes: this is because, for files larger than 4 megabytes, Apache sends the file data in records of 8192 bytes, instead of the maximum record size of 16384.

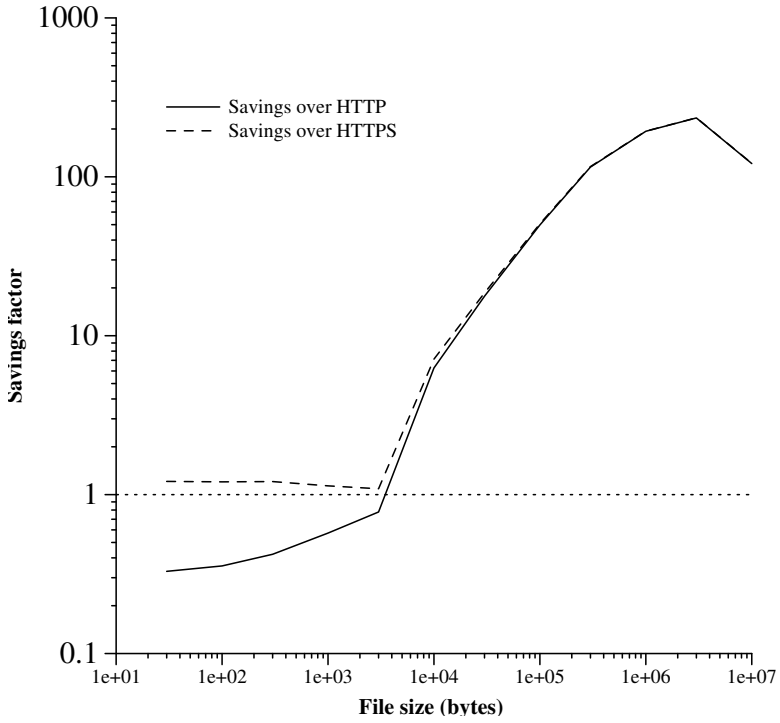

Figure 5-5: Bandwidth savings of SSL splitting over HTTP and HTTPS.

5.3 Latency

When the proxy's cache is cold, SSL splitting performs similar work to regular SSL; thus, we expect their latency characteristics to be similar. However, repetition within the trace confuses the analysis of latency factors, since cached files are faster to download than uncached files. To address this issue, we filtered out repetitions from the `www.lcs` trace chunk, and using the resulting uncacheable trace, measured the start and end times of each request. In order to avoid congestion effects, we performed requests one at a time.

The resulting graph of latencies versus file size is shown in Figure 5-6. It shows three clear lines, one for each of HTTP, HTTPS, and cold SSL splitting. Cold SSL splitting is about 10% faster than HTTPS for small file sizes and about 10% slower than HTTPS for large file sizes, but for the most part they are a close match: the majority of file downloads had less than a 5% difference between the two latencies.

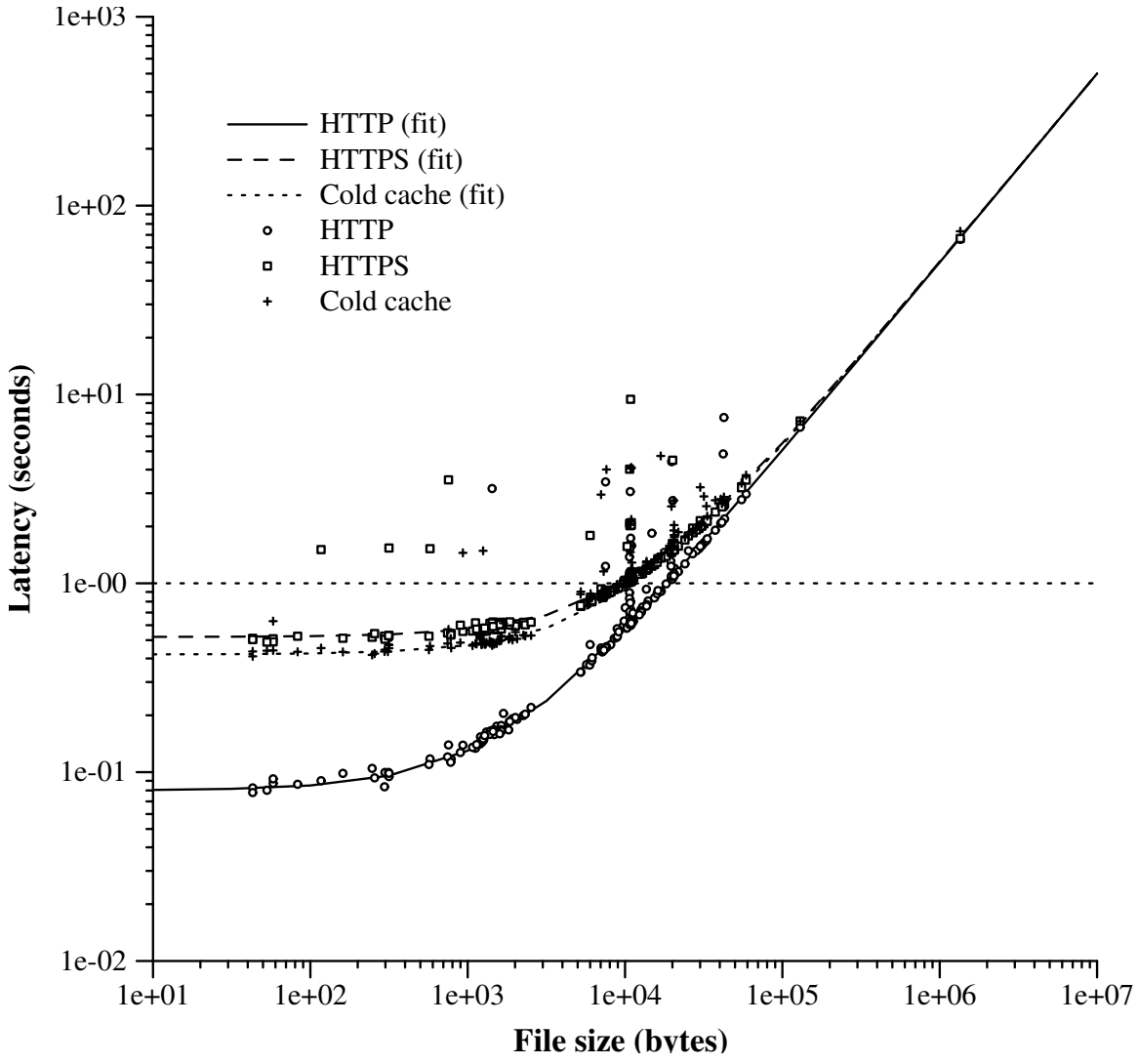

Figure 5-6: Distribution of latencies vs. filesizes.

Chapter 6

Discussion

6.1 The implications of key exposure

The implementation of SSL splitting provides the option to accept clients using a ciphersuite with encryption, and to expose intentionally the encryption key to the proxy (see Section 2.5). Since the only indication of security in most web browsers is a simple on/off lock icon, there is a legitimate question of whether it is reasonable to mislead clients about whether their communications with the server are encrypted and secure against eavesdroppers. We expect that most applications of SSL splitting will not have any Web forms, since it would be pointless to try to cache dynamic content. However, a user browsing an URL beginning with `https:` might reasonably believe that his browsing pattern was not available to eavesdroppers; the only way to deal with this, if it is believed by the server administrator to be a problem, is to notify the user in the text of the Web page. In any case, we should note that SSL splitting does not change the contract presented by SSL in any concrete way: normal SSL provides no guarantee that the server will not broadcast its transmissions to the world.

6.2 Alternative proxy design

In the current design of the SSL splitting protocol, the cache is a transparent forwarding proxy operating at the level of SSL records. An alternative approach would have been to have the client connect directly to the server, and have the server compress its outgoing stream by converting outgoing IP packets containing SSL records to stub records sent to

the cache. The cache would then reconstitute the IP packets and send them to the client, forging the server's IP address and the TCP header. This would have the benefit of reducing the number of round-trip packet flight times from four to three, and would also permit the server to "stripe" a connection with a single client across multiple caches. An additional benefit is that no DNS or HTTP redirection technique would be necessary, since the client's transmissions would be directly to the server. However, operating at the IP layer instead of the SSL record layer is fraught with peril: operating system interfaces are nonstandard and unreliable, networks are likely to black-hole forged packets, and TCP will not behave properly. Also, such an approach risks creating an "open relay" which could be used by malicious clients to hide the source of a denial-of-service attack.

Chapter 7

Related work

Caching and replication in the Web is a subject of much study. Like content-distribution networks [12] and peer-to-peer systems [15], the primary focus of Barnraising is cooperatively sharing the load of serving data. The main difference between Barnraising and previous work is the use of SSL splitting, which allows Barnraising to serve data securely through untrusted proxies to unmodified clients.

7.1 Verifying integrity

The standard approach to providing integrity of data is signing the cryptographic hash of the data with the server's private key (or with the private key of the data's owner). When the client receives the data and its signature (perhaps through different channels), it verifies the integrity of the data by verifying the signature. This solution is typically bundled in the client for a specific application, which users must download to use the application. RPM [22] and FreeNet [2] are among the many applications that use this solution.

The system closest in spirit to Barnraising is read-only SFS [8]. SFSRO allows secure distribution of software bundles through untrusted machines. It provides a generic file system interface, allowing unmodified applications to use SFSRO to distribute data securely. However, SFSRO requires that an SFS client runs on the client machine, which restricts its deployment to SFS users. On the other hand, unlike SSL splitting, the SFSRO server only has to serve the root block to clients, and the computational requirements on the server, untrusted machines, and client are low.

Untrusted surrogates [6] allow storage-limited clients to cache data on nearby surrogate

machines. A server stores data on the surrogate on behalf of the client, and sends the hash of the data to the client; hence, the client can verify the integrity of data when retrieved from the surrogate.

7.2 HTTPS proxies

WASP is a proxy for HTTPS connections [14]. Like SSL splitting, it doesn't require client changes, and defines a separate protocol between proxy and server. Unlike SSL splitting, WASP sends the SSL master secret to the proxy. Since SSL uses the master secret to compute the session keys for both encryption and authentication, this solution puts considerably more trust in the proxy than SSL splitting does. A malicious WASP proxy can change the cached data without the client knowing it.

Proxy certificates [20] provides restricted impersonation within an X.509 public-key infrastructure. A Web site could generate a proxy certificate and hand it to a proxy. The client can then verify the proxy certificate to determine whether the proxy is trusted by the web site to serve the data. Proxy certificates require client changes to process the new X.509 certificate extensions fields.

7.3 Content distribution systems

Commercial content-distribution systems [12] own the machines they use for serving data and therefore trust them. When a client contacts a server with HTTPS via a content-distribution network, the client must trust the content-distribution network to authenticate the server. If SSL splitting were used, the client itself could authenticate the server; also, this would simplify the operation of the content-distribution system.

Most of the content distribution systems based on recently-developed, scalable lookup primitives [18, 13, 3] protect the integrity of data by identifying the data by its cryptographic hash, but the clients must run specialized software to participate in those systems. Squirrel [9] doesn't require special client software, but it doesn't provide data integrity. These systems, however, complement Barnraising by providing it with good techniques for organizing the proxy set.

Chapter 8

Summary

SSL splitting is a novel technique for safely distributing the network load on Web sites to untrusted proxies without requiring modifications to client machines. However, because SSL splitting is only effective at reducing bandwidth consumption when the proxy has access to the plaintext of the connection, it is not appropriate for applications that require confidentiality with respect to the proxy. In addition, SSL splitting incurs a CPU load on the central server due to expensive public-key cryptography operations; it does not address the issue of distributing this load.

The main benefits of SSL splitting is that it provides an end-to-end data-integrity guarantee to unmodified SSL clients, that it reduces the bandwidth consumed by the server, and that it requires only a simple protocol between the server and the proxy. Experiments with a modified OpenSSL library that supports SSL splitting show significant bandwidth savings for files larger than 4000 bytes: when the data of a file is cached on the proxy, the server need only transmit the SSL handshake messages, HTTP header, MAC stream, and payload IDs. Because of these advantages and the ease of deployment, we hope that SSL splitting will form a convenient transition path for content-distribution systems to provide end-to-end data integrity.

Bibliography

[1] B. Cain, A. Barbir, R. Nair, and O. Spatscheck. Known cn request-routing mechanisms. draft-ietf-cdi-known-request-routing-02.txt, Network Working Group, November 2002.

[2] Ian Clarke, Oscar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, California, June 2000. `http://freenet.sourceforge.net`.

[3] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

[4] Tim Dierks and Eric Rescorla. The TLS protocol version 1.1. draft-ietf-tls-rfc2246-bis-02.txt, Network Working Group, October 2002.

[5] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[6] Jason Flinn, Shafeeq Sinnamahidee, and M. Satyanarayanan. Data staging on untrusted surrogates. Technical Report IRP-TR-02-2, Intel Research, May 2002.

[7] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol version 3.0. Internet draft (draft-freier-ssl-version3-02.txt), Network Working Group, November 1996. Work in progress.

[8] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20(1):1–24, February 2002.

[9] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized, peer-to-peer web cache. In *21st ACM Symposium on Principles of Distributed Computing (PODC 2002)*, July 2002.

[10] John Jannotti, David K. Gifford, Kirk L. Johnson, M. Frans Kaashoek, and James W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proc. of the 4th OSDI*, pages 197–212, October 2002.

[11] David Karger, Tom Leighton, Danny Lewin, and Alex Sherman. Web caching with consistent hashing. In *The eighth Word Wide Web Conference*, Toronto, Canada, May 1999.

[12] B. Krishnamurthy, Craig Wills, and Yin Zhang. On the use and performance of content distribution networks. Technical Report TD-52AMHL, ATT Research Labs, August 2001.

[13] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceeedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, pages 190–201, Boston, MA, November 2000.

[14] Nagendra Modadugu and Eu-Jin Goh. The design and implementation of wasp: a wide-area secure proxy. Technical report, Stanford, October 2002. `http://crypto.stanford.edu/~eujin/papers/wasp.ps`.

[15] Any Oram, editor. *Peer-to-peer: Harnessing the power of disruptive technologies*. O'Reilly, March 2001.

[16] E. Rescorla. HTTP over TLS. Rfc 2818, Network Working Group, May 2000.

[17] Eric Rescorla. *SSL and TLS*. Addison-Wesley, 2001.

[18] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, October 2001.

[19] H. Shacham and D. Boneh. Fast-track session establishment for tls. In M. Tripunitara, editor, *Proceedings of NDSS*, pages 195–202. Internet Society, February 2002. `http://hovav.net/`.

[20] S. Tucke, D. Engert, I Foster, V. Welch, M. Thompson, L. Pearlman, and C. Kesselman. Internet x.509 public key infrastructure proxy certificate profile. Internet draft (draft-ietf-pkix-proxy-03), Network Working Group, October 2002. Work in progress.

[21] D. Wessels. *Squid internet object cache*. `http://squid.nlanr.net/Squid/`.

[22] www.rpm.org. *RPM software packaging tool*. `www.rpm.org`.