mSSL: Extending SSL to Support Data Sharing Among Collaborative Clients

Jun Li and Xun Kang
University of Oregon
Department of Computer and Information Science
{lijun, kangxun}@cs.uoregon.edu

Abstract

Client-server applications often do not scale well when a large number of clients access a single server. To solve this, a new trend is to allow a client to download data from other peer clients, in addition to from the server directly. This paradigm, which we call the hybrid peer-to-peer paradigm*, is friendly to the server's scalability, but also faces new security challenges. For example, how can the server authenticate its clients and support data confidentiality? How can a client trust the data downloaded from other clients? What if a client refuses to acknowledge the service it received or overstates the service it offered?*

In this paper, we present a protocol, called mSSL*, that provides a set of security functions to enable secure sharing of the data of a server among its clients. In addition to access control and confidentiality support, mSSL provides an original design on supporting data integrity and proof of service in this new context. Our evaluation further shows that mSSL has a reasonable overhead.*

1 Introduction

While conventionally a client needs to directly request data from a server, a new trend is to allow multiple clients, such as thousands of clients of a web server, to share data among themselves in a peer-to-peer fashion [3, 21, 8, 20]. This creates a *hybrid peer-to-peer paradigm* that involves both client-server and peer-to-peer communications. If a client has downloaded some blocks of data from a server, other clients can obtain those blocks from this client, rather than the original server. This mechanism can potentially prevent a server from being overloaded when serving a large number of clients, and enable even an under-provisioned site to provide a scalable data service.

Accompanying this trend, however, are new security challenges that conventional client-server approaches such as SSL [17] cannot easily address. Especially, how can a server allow peer-to-peer data sharing without weakening client authentication and access control? When a client retrieves data, whether from a server or its peer clients, can the integrity of the data always be guaranteed? How can data confidentiality be supported? And, if a client provides data to another client, can the former provide a non-repudiable proof of its data provision service? Such questions can be easily raised if the hybrid peer-to-peer paradigm is used to distribute software packages, sell large multimedia files, share critical information among participants, or other applications that face various security issues.

Corresponding to the conventional SSL service that protects client-server connections, we propose to build a *mSSL* service that protects both client-server and client-client communications in the hybrid peer-to-peer paradigm. mSSL provides a set of functions that enable secure sharing of a server's data among its clients and support applications that sit on top of mSSL. Both conventional security issues such as client authentication, data integrity, and data confidentiality and new security issues such as proof of service will all be addressed in this new context. Our solutions to data integrity and proof of service are especially new.

One important goal of mSSL is that, while allowing clients to share data traditionally downloaded directly from a server, the security should not be weakened compared to the traditional client-server model, or only weakened to a minimal degree if at all. Unlike a client-server environment, a server does not directly control all the data flow. A client may have to decide whether to provide data to another client, or may need to verify the data received from another. Also unlike a pure peer-to-peer environment, security solutions for the hybrid peer-to-peer paradigm can take advantage of the existence of a server, enabling a potential integrated "centralized plus distributed" approach.

We have also evaluated the performance of mSSL. The security functionalities provided by mSSL may be used in different combinations for different applications, and the success of mSSL will depend on its efficacy under all those combinations. Of particular concern is the overhead introduced by mSSL and how much an application could be slowed down by mSSL.

The rest of this paper is organized as follows. We illustrate the design of mSSL in Sections 2, 3 and 4, focusing on the major security functionalities mSSL provides. We then discuss several key issues in Section 5. In Section 6 we evaluate the performance of mSSL. Section 7 describes the related work. We conclude the paper in Section 8.

2 Design

The goal of mSSL is to provide a light-weight, scalable, robust, and flexible security protocol to provide a suite of security functionalities for applications running using the hybrid peer-to-peer paradigm. mSSL can also be used as a library of function calls or regarded as a middleware service. These security functionalities include: (1) *client authentication* to ensure that only authenticated clients can obtain a server's data, whether or not directly from the server, (2) *data integrity* to protect the integrity of the data whether the data is received directly from the server or indirectly from other clients, (3) *data confidentiality* to encrypt data for confidentiality, and (4) *proof of service* to allow a client to prove to the server that it has provided specific data-sharing service to other clients.

mSSL supports two different access modes for a client to obtain its server's data: *direct access* and *indirect access*. In both modes, the client will first create an SSL secure channel between itself and the server and then authenticate itself (for instance using its account name and password or using an identity certificate). If in the direct access mode, the client will then directly receive a copy of the requested data from the server; but if in the indirect access mode, the client will obtain necessary information from the server and then turn to other peer clients to receive the data. mSSL in direct access mode is essentially the same as SSL, and it is mSSL in indirect access mode that we will focus on.

In the following, we call a client who provides data to others a *provider*. A client who receives data from provider clients is called a *recipient*. We also interchangeably use the terms *data*, *data object*, and *file*.

In the rest of this section, we describe the straightforward design in mSSL for client authentication and confidentiality. Then in Sections 3 and 4, we illustrate our design for data integrity and proof of service, the two security functionalities that highlight our contributions for security in this hybrid peer-to-peer paradigm.

2.1 Client Authentication

mSSL implements a ticket-based solution to ensure that only authenticated clients may access a data object, no matter where the data object is located. In both direct and indirect modes, a client must first contact its server and authenticate itself through a SSL channel. Once the server decides that the client is allowed to obtain the data (such as after a client paid for purchasing an audio file), the server will then either directly transfer the data to the client, or provide a ticket for the client to contact other peer clients.

The ticket proves that the server authorizes this particular client to download data from other clients, who can verify the ticket and authenticate the client in question before providing data to this client. The ticket typically includes the id of the client requesting a data object, the id of the data object, the time that the ticket is issued, the validity period of the ticket, and a sequence number. The server also uses its private key to sign the ticket so that any provider that knows the server's public key can verify the ticket.

2.2 Confidentiality

In order to ensure that only authenticated clients can access data, confidentiality is necessary. mSSL adopts an object-key-based approach. Every data object can be associated with an object key for encrypting or decrypting the data object. An object key can have a life time and be replaced when it expires. Essentially an object-oriented approach, this scheme is able to enforce a fine-grained access control at the data object level. A server can encrypt any data object just once in advance for all potential clients, instead of once per client.

When a client c requests a data object O in its encrypted form $k_o\{O\}$ from a server S, S can issue O's object key k_o to c immediately after authenticating c's request (or after c finishes downloading $k_o\{O\}$). After c receives $k_o\{O\}$ from either c's peers or S, c can use k_o to decrypt $k_o\{O\}$. Note that the distribution of k_o is protected by k_c, a secret key established between c and S over their SSL channel. Figure 1 shows this procedure in indirect access mode.

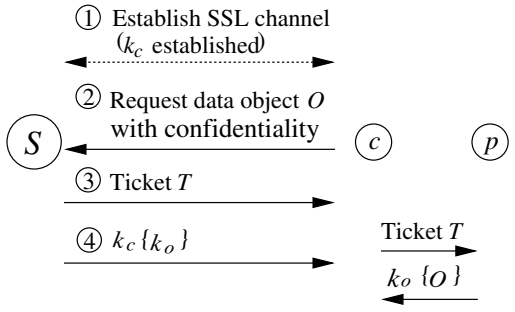

Figure 1. Confidentiality through mSSL (indirect access mode)

3 Integrity

We now discuss how mSSL allows a client to verify the integrity of its server's data, whether the data is directly received from the server or indirectly from some providers.

A simple solution is to ask the server to sign the whole data object. Unfortunately, if the signature validation fails, the *whole* data object must be retransmitted. When the data object is large, this can be a serious problem. Instead, mSSL allows the client to verify the integrity at the block level, assuming every data object can be divided into blocks. A client does not have to wait until it receives all the blocks of a data object before verifying the integrity. If it detects an invalid block, it can request a retransmission of that single block immediately.

In the following, after presenting two straightforward but flawed approaches, we show how Merkle hash tree can be used in a new way to provide mSSL's block-based integrity solution.

3.1 Straightforward Approaches and Their Drawbacks

One approach is to bundle a signature with every block of an object. A server signs every block by encrypting a strong one-way hash of the block. To determine the integrity of a newly received block, a client checks whether decrypting the block signature would lead to the correct hash of the block. However, encryption and decryption per block will lead to a high computational overhead.

Another approach is to build a signed superblock for a data object. It contains the hash value of every block of the object and a signature of the whole superblock. The superblock will be transmitted first before transmitting data blocks. After a client receives an authentic superblock of a data object, it then uses the hash values from the superblock to verify the integrity of each block that it later receives.

However, a superblock can be very large, leading to a high startup latency. For a data object with 2^m blocks, assuming every block's hash value is 16 bytes (all modern hash algorithms produce hash values of 16 bytes and higher) and the superblock's signature is also 16 bytes, its superblock will be $(16 * 2^m + 16)$ bytes. A $1GB$ file with $1KB$ block size will have a superblock with $(16M + 16)$ bytes. As the size of a data object increases, the size of its superblock increases linearly. A large superblock can cause a significant delay before a client receives the very first block, which would not be acceptable to applications in which users prefer prompt response, such as multimedia streaming. Moreover, if the superblock itself is corrupted, the retransmission can also be costly. Note that although increasing the block size can reduce the size of a superblock, the retransmission cost of individual blocks will increase.

3.2 The Merkle Hash Tree of a Data Object

We assume a data object O is divided into 2^m blocks, and its binary Merkle hash tree is $M(O)$. The height of $M(O)$ is then $m + 1$, with the root at level 0 and leaf nodes at level m. Figure 2 shows an example tree.

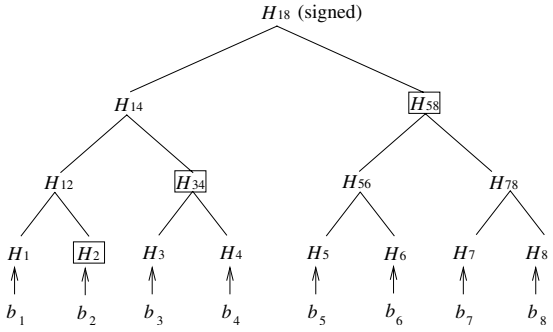

Figure 2. Merkle hash tree of a data object with 8 blocks. The authentication path of block b_1 is $< H_2, H_{34}, H_{58} >$.

We first introduce notations for representing a node or its value on $M(O)$:

- H^l: A node at level l.
- H_i^l: The node at level l that is the ith node counting from left. (Note the first node is H_1^l)
- H_i: The ith leaf node counting from the left, *i.e.* H_i^m.
- H_{ab}: The node which is the root of the subtree containing leaf nodes H_a through H_b (inclusive).

We can calculate $M(O)$ from the bottom up using a one-way hash function f as follows: (1) *Leaf nodes*. For every block b_i $(i = 1, ..., 2^m)$, $H_i = f(b_i)$. (2) *Non-leaf nodes*. For every two sibling nodes H_{2i-1}^l and H_{2i}^l, the value of their parent H_i^{l-1} is $f(H_{2i-1}^l, H_{2i}^l)$. (If a node H_{2i-1}^l has no sibling, its parent H_i^{l-1} is H_{2i-1}^l.) Applying the parent calculation process recursively, we will obtain the value of every non-leaf node, including H^0, the value of the root node. (3) *root*. H^0 is signed by the server, but note that all other hash values are *not* signed.

3.3 Conventional Integrity Verification Based on The Merkle Hash Tree of a Data Object

To verify the integrity of every block of the data object O, a client will first request the signed value of tree $M(O)$'s root, H^0. Once it receives a block b, the client will also request to receive b's *authentication path* $A(b) = \langle H^m, H^{m-1}, ..., H^1 \rangle$. $A(b)$ contains exactly one particular hash value from every level of $M(O)$, where H^{i-1} is

the sibling of H^i's parent. H^{i-1} is also called H^i's *uncle*. With $A(b)$, the procedure in Section 3.2 can be used to calculate H^0. If block b is modified (or any values on $A(b)$ is modified), the calculated H^0 will not equal the signed root H^0, leading to the detection of an integrity violation. The client can then request the retransmission of block b.

With this solution, a client does not have to download all the hash values beforehand as in the superblock-based solution, nor does it need to perform expensive encryption or decryption operations. However, this solution can lead to a high traffic overhead. For a data object with 2^m blocks, every block's authentication path will have m hash values. Assuming each hash value is 16 bytes, the total amount of overhead traffic will then be $16m * 2^m$ bytes, $\frac{16m}{|b|}$ of the data traffic ($|b|$ is the number of bytes of a block).

3.4 mSSL's On-demand Data Integrity Solution via Integrity Path

mSSL's integrity solution optimizes both the on-demand requests of integrity verification information, and the verification procedure. In fact, when a client needs to obtain an authentication path $A(b)$ of a newly downloaded block b, it may not need to download every hash value $h \in A(b)$. For a given hash value $h \in A(b)$, if $h \in A(b')$ where b' is another block and has been verified earlier, h must be already available locally. For example, in Figure 2, the authentication path of block b_1 and b_2 are $A_1 = \langle H_2, H_{34}, H_{58} \rangle$ and $A_2 = \langle H_1, H_{34}, H_{58} \rangle$, respectively. If a client received b_1 first and then verified its integrity by obtaining A_1, it does not need to download H_{34} and H_{58} again when verifying b_2. Moreover, certain hash values will have to be calculated during the local block integrity verification process, and can become available for verification along other authentication paths. Using the same example, when verifying b_1's integrity, H_1 (as well as H_{12} and H_{14}) will be calculated. Therefore, when verifying b_2 along A_2, there is no need to download H_1 either.

In contrast to the authentication path concept, we now introduce a concept called *mSSL integrity path* for optimized block integrity verification. A block's mSSL integrity path consists of all those hash values from this block's authentication path that are *not* locally available. To determine what are these hash values, we introduce Theorem 1 below:

Theorem 1 If h is locally available at a recipient client r, then h's uncle $uncle(h)$ is also locally available.

We omit the proof of Theorem 1 to save space. Now, suppose a block b's authentication path $A(b)$ is $A(b) = \langle H^m, H^{m-1}, ..., H^1 \rangle$, and denote b's mSSL integrity path $mip(b)$. If H^{l-1} is locally available, but H^l is not, then according to Theorem 1, we know $H^{l-1}, H^l, \cdots, H^1$ are all locally available, and none of $H^m, H^{m-1}, \cdots, H^l$ are. Thus, $mip(b) = \langle H^m, H^{m-1}, \cdots, H^l \rangle$, with a total of $|mip(b)| = m - l + 1$ hash values. $|mip(b)|$ is also the number of levels needed to be downloaded in order to have a complete authentication path.

The integrity path concept greatly simplifies the on-demand request process. When a client receives block b from a server or a provider, it can determine $mip(b)$ first and then send a request for $mip(b)$ to the server or the provider. Notice that it is sufficient for the request to just contain the value of $|mip(b)|$ since the provider or the server can easily determine what $mip(b)$ is. Figure 3 describes this procedure, where a client maintains an object O's Merkle hash tree, $t_c(O)$, that only keeps locally available hash values.

Once the client receives $mip(b)$, it could further derive $A(b)$ and apply the procedure from Section 3.3 to verify the integrity of block b. However, mSSL also optimizes this verification process. Since H^{l-1} was locally available, an earlier integrity verification that involves H^{l-1} must have used $sibling(H^{l-1})$ to calculate all of H^{l-1}'s ancestors (including the root), therefore an authentic $sibling(H^{l-1})$ is also locally available ($sibling(H^{l-1})$ is also H^l's parent). As a result, the verification of block b actually only needs to compare a newly calculated $sibling(H^{l-1})$ with the current $sibling(H^{l-1})$ to decide if block b is integral; the additional calculations from level $l-1$ up would be repeated calculations.

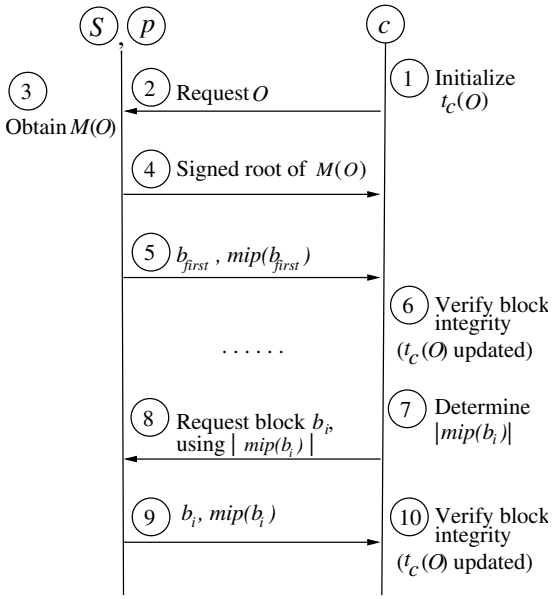

Figure 3. mSSL's integrity solution. Except for the very first block b_{first}, the client c requests on-demand the integrity path of every block.

4 Proof of Service

4.1 Overview

An obstacle to realizing the benefits of data sharing among peer clients is whether clients have incentives to share data. To address this, we assume a simple but generic economy model: (1) A provider will receive credits for assisting its server. (2) A recipient will pay less for the data it receives from providers since it does not directly utilize as much of its server's resources. (3) By offloading some tasks to its providers, a server will serve more clients overall and thus make more profit, even though it charges each individual client less.

However, in order to harden such an economy model, it is critical that a trustworthy, effective proof-of-service mechanism must be designed. With proof of service, a client can present to its server a proof of its service to others, a server can verify whether or not a client has indeed served other clients as it claims, and a recipient cannot cheat or be cheated about its reception of data from others.

We describe our approach to providing proofs of service below. This approach must ensure trustworthiness of proofs. Furthermore, given that a large number of clients of a single server may be involved, the approach must be scalable; in particular, the server should not be overloaded and the traffic and storage overhead should not be high.

Our basic solution is to assume that every data object is divided into multiple blocks and enforce an interlocking block-by-block verification mechanism between every pair of provider and recipient. For every block that a provider has sent to a recipient, the recipient will verify the integrity of the block and sends back a *non-repudiable* acknowledgment to the provider. The provider will verify the acknowledgment *before* providing the next block. Those verified acknowledgments can then be used as the proof of the service that the provider has offered to other clients.

However, problems may arise in this basic solution. Clearly, if a provider has to present a separate acknowledgment as the proof for *every* block it served, there will be a proof explosion with large files. Also, after receiving a block from a provider, a recipient may run away without acknowledging the receipt of this block, leaving the provider unable to have a proof for serving the block. Finally, the server can be overloaded if clients require the server to compose or verify acknowledgments frequently.

We address these problems. Figure 4 shows the proof-of-service operations between a provider p and a recipient r. Basically, in steps 1 and 2, p and r will go through a handshake, including exchanging public key certificates. Then after r requests block b_i (step 3), p will send block b_i encrypted using a secret *block key* to r in step 4. In step 5, r acknowledges its receipts of the encrypted b_i and, if needed, requests another block. p will verify the acknowledgment in step 6; if verified, in step 7 it sends the block key in its protected form to r, which can then obtain the block key, use it to decrypt the encrypted block, and verify the integrity of the block.

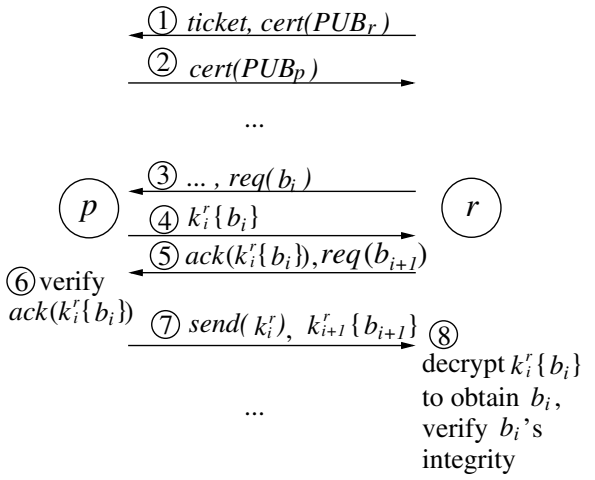

Figure 4. mSSL's proof-of-service solution

4.2 Step-wise Explanations and Justifications

In the following, we explain and justify our steps above.

Steps 1 and 2 show the exchange of public key certificates between p and r. We first make sure that every client has a certified public key. This is important because clients in the hybrid peer-to-peer paradigm are often ordinary users and do not have certified public keys. Noticing that while running the SSL protocol, the server itself has a certificate for its own public key, we use the server as a public key CA (certificate authority) to certify the public keys of its own clients. A client needs to generate beforehand a key pair by itself using a public key generation algorithm, then send a certificate request to the server. The server in turn will generate a certificate that is signed with the server's private key and send it back to the client. As a result, every client can have its own public key certified by the server, and every client is also able to verify the certificate of its peers.

As shown in step 4, a secret *block key* is also used for encrypting a block. When the provider sends the ith block b_i from a data object oid to a recipient rid, it will generate a block key k_i^r as:

$$k_i^r = f(pid, rid, oid, i, k_p) \quad (1)$$

where f is a one-way hash function and k_p is the secret key shared between the provider and the server. Note the server can also apply this formula to calculate the block key.

For acknowledging an encrypted block $k_i^r\{b_i\}$, in step 5 the recipient forms the acknowledgment $ack(k_i^r\{b_i\})$:

$$PRV_r\{pid,\ rid,\ oid,\ sack,\ timestamp,\ d(k_i^r\{b_i\})\} \quad (2)$$

where PRV_r is the private key of r for signing the acknowledgment (so that r cannot deny it later), $timestamp$ records when the acknowledgment is issued, $d(k_i^r\{b_i\})$ is the digest of the encrypted block using a one-way hash function. Furthermore, the $sack$ field in acknowledgments solves the proof explosion problem. It is in a format similar to the SACK options for the TCP protocol [14], and can express *all* the blocks that r has received from p, instead of just the most recent one. For example, it can be $[0-56, 58-99]$ to confirm the reception of the first 100 blocks of a data object except for block 57.

In step 6, p will verify the acknowledgment to decide whether or not to provide the block key to the recipient in step 7. This includes verifying the digest field $d(k_i^r\{b_i\})$ so that when p presents S this acknowledgment as the proof of its service, S can verify whether r received the correctly encrypted last block, i.e., correct block b_i encrypted using correct block key k_i^r. Note that the server can use Equation (1) to calculate k_i^r.

In step 7, $send(k_i^r)$ is the delivery of the block key in a protected form:

$$PRV_p\{pid, rid, oid, i, PUB_r\{k_i^r\}\} \quad (3)$$

where PRV_p is p's private key, PUB_r is r's public key, and obviously r—and only r—can use PUB_p (p's public key) and its own private key to obtain k_i^r. Also, if r cannot decrypt $k_i^r\{b_i\}$, it can forward the protected block key to S so that S can verify if p sent a bad block key.

In case p did not or could not provide a block key at all (i.e. no step 7 happening), r can retrieve it by asking S to apply Equation (1) to calculate the block key. When doing so, r must send S an acknowledgment as in Equation (2) so that S knows r has indeed received a correctly encrypted block from p. Also, the server expects that there is only one such query for each recipient-provider pair, since the occurrence of this query means that after this query r would decide that p should not be relied upon any more.

Finally, in step 8, after the recipient receives the block key, it can decrypt $k_i^r\{b_i\}$ (received in step 4) to obtain b_i. Moreover, it can verify the integrity of b_i. Only if b_i is integral will the recipient continue with the provider for the next block. In case b_i appears to be corrupted, r knows that p cannot present $ack(k_i^r\{b_i\})$ to S as a correct proof of its service—S knows the correct b_i and k_i^r.

4.3 Performance and Scalability Considerations

The performance of the proof-of-service protocol can be further accelerated. We introduce parallelism in handling multiple blocks concurrently. We require every acknowledgment to include digests of the last m encrypted blocks, and the server will verify whether the recipient received the correctly encrypted blocks for last m blocks (instead of just last one block). Meanwhile, upon the receipt of block key k_i^r and the encrypted block b_{i+1} (i.e. $k_{i+1}^r\{b_{i+1}\}$), r will first acknowledge the receipt of b_{i+1} *before* proceeding to step 8. With this design, it can repeat this prompt acknowledgment process for the next $m-1$ blocks, greatly improving the performance. In case that block b_i is discovered corrupted, the provider will still not able to have a proof that it successfully delivered b_i—since the proof must show correct digests of all last m blocks. Here, we want to select m such that it is small to be scalable, but large enough to keep high level of parallelism.

Throughout the whole design, the server's load has been kept very light. A provider can wait until the end of serving a recipient to present a single proof of its service toward this client. Also, the only type of query that a recipient can issue is to verify or retrieve the block key of a block it receives from a provider (see discussion on step 7 above), which happens only once per recipient-provider pair.

5 Discussion

mSSL is configurable to support different needs. In this section, we first discuss how different security functionalities of mSSL can be combined. We then look at possible attacks against mSSL and suitable countermeasures. Finally, we look at the limits of mSSL.

5.1 Combining Security Functionalities

Using A, C, I, and P to represent the four primary security functions that mSSL provides—client authentication, confidentiality, integrity, and proof of service, below we show their totally *six* meaningful combinations.

Considering a data object O at a server S, we first look at requiring just one of the four:

- A: Some clients are allowed to access O, some are not. A client thus must authenticate itself.
- I: Every client can obtain O, but needs to ensure the integrity of O. Here, S offers O for everyone, so no confidentiality or client authentication are needed.
- C: Confidentiality of O is required. If so, A must also be utilized to make sure the encrypted O is delivered only to authorized clients. So C implies AC.
- P: Proof of service is required. Note P must be combined with A: a recipient needs a certificate for its public key, thus it must first authenticate itself; the server

must also authenticate a provider before it can verify the provider's proof of its service to others. P must also be provided together with I: in the proof of service design, a recipient needs to verify the integrity of a received block. On the other hand, P includes its own method of encryption, thus C is not needed. Therefore, enforcing P implies enforcing A, I, and P all together, i.e. AIP, but not C.

Now we consider possible combinations of two of the four functions. A two-function combination cannot have P, since P implies AIP. C already implies AC. So the only new combination possible is AI. Furthermore, the only new combination with three of the four primaries is AIC. We will not have a combination of all four, $AICP$, since P implies not using C.

Therefore, we have totally six scenarios: A, I, C (i.e. AC), P (i.e. AIP), AI, and AIC.

Every mSSL application may have its own needs and thus choose a specific scenario. (Note that except for the I scenario, every scenario requires A, meaning every client must authenticate itself to its server.) For example, the I scenario can be useful when a server is providing critical public information and all that is needed is to guarantee the integrity of the data. The AIC scenario may be chosen when encrypted data sharing using C is not sufficient; sometimes even if a recipient can decrypt data from a provider successfully, it may not trust the provider and still want to verify the integrity to make sure the data is indeed unaltered from the server's original.

5.2 Preventing Attacks

The integrity solution of mSSL can be invoked to ensure that every block of the data object is authentic, even without authentication invoked at the server. Since every client can obtain a signed root value of the Merkle hash tree of the data object, and the authentication path of every block, any modification of the block by anyone will cause the block to fail the integrity verification process. Moreover, mSSL allows a data object to be transmitted in its encrypted form to provide confidentiality. The attack against the proof-of-service scheme is probably the most complicated; we focus on this attack in the following.

Generally speaking, two types of proof-of-service attacks may occur: individual cheating and colluded cheating. An individual client may cheat by misreporting the amount of services it has received or provided. Or, multiple clients may work in collusion, forging false proofs of service together. Both types are for the purpose of obtaining non-deserved credits from the server.

- **Individual Cheating:** An individual client can cheat in the following ways: (1) A provider overstates its service for extra credit from the server. It claims that it sent another client certain data although it did not. (2) A recipient deliberately refuses to acknowledge its receipt of specific data blocks from another provider, so that the provider cannot show the proof of serving those blocks.

  The proof-of-service design in mSSL addresses both methods of cheating above. For the first type of cheating, from Section 4 we know that every proof is a signed acknowledgment from a recipient. A provider does not know the key other clients use to sign acknowledgments, so it cannot forge any. For the second type of cheating, note that if a recipient denies its reception of a data block, it will not generate an acknowledgment for the block; without the acknowledgment, it will not be able to receive the secret block key to decrypt the block. So this cheating is also not possible.

- **Colluded Cheating:** In order to obtain extra non-deserved credits, multiple clients may collude to forge proofs of services that did not actually take place. We list several cases here: (1) A provider forges a proof that it provided itself certain data. (2) While a recipient r sends acknowledgments to its real provider p_1, it also sends a copy of every acknowledgment to its accomplices a_1, a_2,..., a_n. Or, r can just copy the acknowledgment that confirms the largest number of received blocks. Then a_1, a_2,..., a_n can all claim that they delivered certain data blocks to r, even though they did not at all. (3) Two clients r and p collude to forge a proof that p provided certain data to r.

  To counteract cases (1) and (2), note the proof-of-service design of mSSL allows a server to determine whether a proof is indeed about the service from a specific client p to another specific client r. A proof must be signed by the recipient, and it contains the digest of last one or several encrypted blocks that a provider has served (Equation 2), and the encryption uses a key that is specific to the provider ((Equation 1). In case (1), the server will find out that it is a proof of self-service; in case (2), the server will detect that only the proof from p_1 is trustworthy.

  Case (3) is the most difficult since the server cannot tell whether or not a service has occurred. If a recipient must pay for every copy of data it receives, even including redundant copies, an economical countermeasure can be designed. This way, while the colluding provider p may gain undeserved credit, the colluding recipient r will be penalized since it will be charged for the forged service.

5.3 The limit of mSSL

When a server and a client establish a SSL channel, they can interact with each other through the channel, including allowing the client to provide certain information to the

server. Not necessarily for mSSL: mSSL is designed not for two-way data transfer, but for securely and efficiently sharing a server's data among peer clients. Note that mSSL does not conflict with SSL, and an mSSL application can have its clients use SSL to interact with a server, and use mSSL to have those clients share data.

Certain security issues that already exist in the traditional client-server model are not addressed by mSSL. For instance, SSL does not guarantee that a client that receives data from a SSL channel will not divulge the data to others (this issue could be potentially addressed by various digital rights management approaches). Neither does mSSL whereas it aims to ensure that security is not weakened compared to the client-server model.

In addition to the security functionalities presented in this paper, other functionalities may also be needed. For example, a security function that prevents traffic analysis may be necessary if users are concerned that an attacker may know that the same encrypted object is being transferred between clients during different sessions. mSSL is extensible and more functionalities can be added in the future.

6 Evaluation

6.1 Overview—Goal, Metrics, and Methods

The goal of measuring mSSL is to obtain the overhead of using mSSL in different scenarios to see whether the overhead of mSSL is acceptable overall, and to compare the overhead among different scenarios.

The metrics we used to evaluate mSSL include:

- *Server capacity:* The number of client requests a server can successfully process per time unit.
- *File downloading time:* The latency from the time a client initiates a connection with a server to the time that the client receives the whole file.
- *Storage overhead:* The space a client or a server needs in order to store mSSL-related information.
- *Control traffic volume:* The volume of control traffic when mSSL is in place.

For every metric, we evaluate mSSL under one of the six scenarios introduced in Section 5.1, plus a *none* scenario in which no security functionalities are provided except that the mSSL infrastructure is in place. The *none* scenario is useful to measure the overhead of the mSSL framework. We thus have a total of seven scenarios for evaluating mSSL: $none$, A, I, C (i.e. AC), P (i.e. AIP), AI, and AIC.

We have implemented mSSL in Java and tested it in our laboratory. Furthermore, we have also implemented a file-sharing application which can invoke mSSL for its security needs. With this application, a client can either directly download a file from a server, or download it from providers that have a copy of the file. The client can ask the server for a list of provider clients, and can become a provider after it obtains the file. We adopted 3DES (112-bit key length) for classical cryptography, RSA (1024-bit key length) for public key cryptography, and MD5 for hashing algorithms.

When mSSL is in direct mode, it will essentially incur the same cost as SSL. Our evaluation of mSSL will focus on its indirect access mode when a client needs to download data objects from provider clients. We use experiments to collect results and perform analysis for server capacity and file downloading time, and analyze the storage overhead and control traffic volume through calculations.

6.2 Server Capacity

We measured the server capacity as following. We connected a client machine (a machine running Linux 2.4.20-20.9smp with dual 2.4 GHz Pentium IV processors and 1 GB memory) and a server machine (an iBook running OSX with a 700 MHz processor and 384 MB memory) through a 100 Mbps link. Then for every one of seven scenarios to study, we had the client flood the server with file downloading requests (each with the same security requirements), and recorded how many such requests the server was able to serve over an extended period. To make sure the server reached full capacity, we made sure that the client machine was lightly loaded (note that the client machine is much more powerful than the server machine), the bandwidth usage between the two was far from saturated, and the server's CPU usage reached 100%.

Figure 5 shows the results. We can see that introducing new security functionalities does invite extra overhead; however, such overhead is acceptable. For example, while under the $none$ scenario, the server capacity averages 465 requests per minute. Adding integrity does not affect the

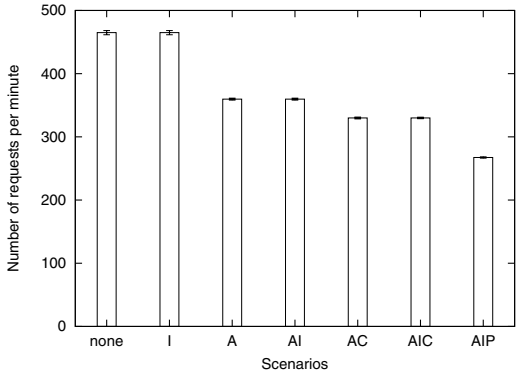

Figure 5. Server capacity

server capacity since no extra data-integrity-related operations are needed at the server. If requiring client authentication, the server capacity will become approximately 22% less. Server capacity decreases another 6% if also requiring confidentiality. Finally, if proof of service is also required, server capacity will decrease another 13%.

6.3 File Downloading Time

In measuring mSSL's impact on downloading files from one or multiple providers, the server is the same machine as in Section 6.2, and every client is a Dell Latitude D810 machine running Linux 2.6.9-ck3 with a 1.73 GHz Pentium M processor and 512 MB memory, all connected over a 100 Mbps Ethernet. We have found that increasing the number of providers of a recipient will linearly increase the downloading speed. In the following, we report the measurement results for the case with one provider, and compare the impact under all different scenarios.

File downloading time includes a *startup latency*, which is the time that a client spends in handshaking with the server before sending out a request for the first block of a file, and *data transferring time*, which is the rest of file downloading time. We report the results for each.

Our measurements show that the startup latency is not related to the size of a file, but different scenarios will have different startup latency. Figure 6 shows that compared to the $none$ scenario, further requiring client authentication, integrity, confidentiality, or their combinations, will not cause a measurable increase of the startup latency, but requiring proof of service will lengthen the startup latency more significantly.

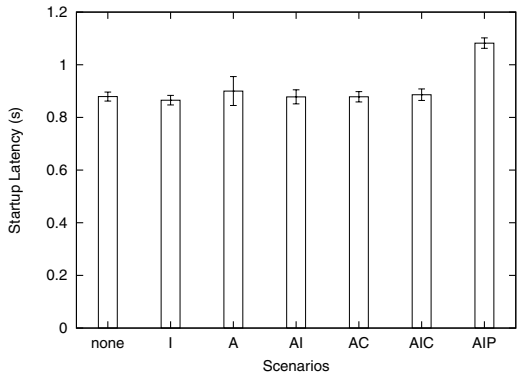

Figure 6. Startup latency

Unsurprisingly, our measurements also show that data transferring time is proportional to the size of a file. In the following, we focus on the difference of data transferring time between different scenarios.

Figure 7 shows how each *basic* scenario (which consists of a single mSSL security function) may slow down the downloading speed. The impact from adding client authentication and/or confidentiality is almost negligible; except for a few extra operations, transferring encrypted data or non-encrypted data does not make a difference in terms of transferring time, and data encryption and decryption operations can be done off-line. There is a slight decrease of downloading speed when adding integrity verification. Recall that a recipient needs to request integrity path information (Section 3). The proof of service function has a more significant impact; for example, with this function turned on, downloading a $32MB$ file will require 63 seconds compared to the 45 seconds needed in the $none$ scenario.

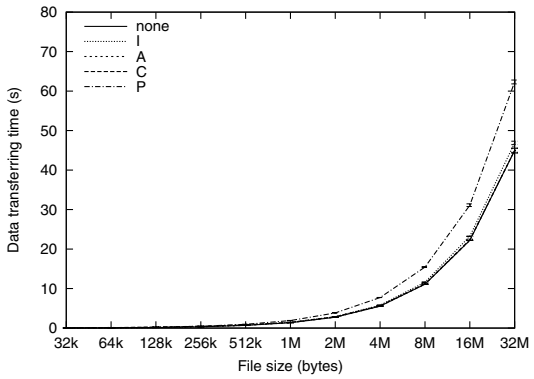

Figure 7. Data transferring time under basic scenarios

Figure 8 shows the data transferring time when supporting all different *combined* mSSL security functionalities. The AIP scenario (i.e. P) is the slowest, and AC is the fastest, while AI or AIC (equivalent because of offline encryption/decryption) are just slightly slower than AC. The reasons for the speed differences are similar to above.

6.4 Storage Overhead

While the $none$ scenario will incur the same storage overhead as in the conventional SSL, more security functionalities lead to extra storage overhead. Assuming the server is S and a recipient r is obtaining a data object O from a provider p, extra storage overhead related to each security function is:

- Client authentication: r needs to store a ticket for O. In our implementation, a ticket is approximately 160 bytes.
- Integrity: Both p and r need to store the Merkle hash tree of O. If O is 1 GB, each block is 8 KB, and every hash value is 16 bytes, the tree will be approximately 4 MB.

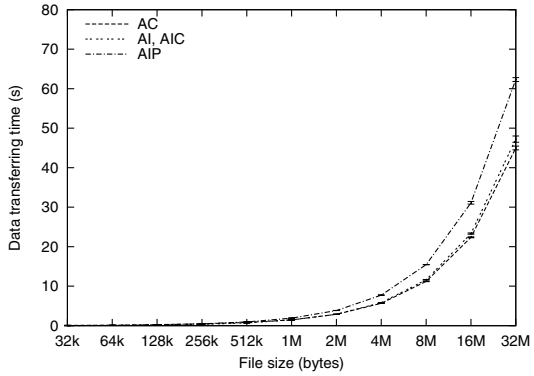

Figure 8. Data transferring time under combined scenarios

- Confidentiality: p needs to store an encrypted copy of O (unless p encrypts O on the fly) and S needs to store the decryption key.
- Proof of service: p and r need to store certificates of their public keys, and p also needs to store necessary acknowledgments from r. Due to the aggregation feature built into the acknowledgment mechanism, only a small number of acknowledgments are needed and each acknowledgment in our implementation is typically 150-200 bytes.

6.5 Volume of Control Traffic

Similarly, while the $none$ scenario will incur the same volume of control traffic as in the conventional SSL, more security functionalities lead to extra control traffic. Again, assuming the server is S and a recipient r is obtaining a data object O from a provider p, extra traffic purely related to each of the following is:

- Client authentication: the delivery of a ticket from S to r and from r to p.
- Integrity: assuming O has n blocks, the extra traffic will be $n-1$ hash values from p to r (we omit the proof to save space) and a small amount of request traffic from r to p.
- Confidentiality: S needs forward the decryption key to r.
- Proof of service: p and r need to send each other a certificate of their own public keys. r also needs to send an acknowledgment for each encrypted block of O. p also needs to send a protected block key to r (see Section 4). p may also contact S to present the proof of its service, which will be the size of an acknowledgment (recall it is typically 150-200 bytes in our implementation). r may also burden S with a small amount of traffic when r has trouble with block keys.

6.6 Summary

Our comprehensive cost and performance study of mSSL shows that in general, more security functionalities lead to higher storage and traffic overhead while decreasing server capacity and lengthening file downloading time. But overall, the extra overhead it introduces is at a reasonable level and generally very small.

7 Related Work

Works related to mSSL can be categorized into related security protocols, related data integrity solutions, and incentive mechanisms in peer-to-peer environment.

7.1 Security Protocols

We discuss security protocols that could potentially be used for the hybrid peer-to-peer paradigm to support functions that mSSL is designed for.

SSL/TLS: SSL [17], or SSL/TLS, provides data encryption and authentication between a client and a server. It is the most common security scheme today for securing web-based services and has also been used for many other services. However, SSL is designed to secure point-to-point communications. To use SSL in securing the sharing of data from a server among its clients, it must be applied separately to every client-server and client-client connection, resulting in a high overhead.

Kerberos: mSSL's ticket-based authentication mechanism has some similarities with Kerberos. Kerberos allows a client to contact a trusted third party, a Key Distribution Center (KDC), to obtain a ticket-granting ticket (TGT), and then use the TGT to obtain a ticket related to a particular service. However, designed for this hybrid P2P environment, mSSL avoids the reliance on a trusted third party. It allows the server to issue a ticket directly to a client.

Group Management: If all authenticated clients of a server are treated as a group, some group management techniques could be useful. For example, SDSI [18] uses a simple PKI to manage memberships and secret communication among members. Various group key management schemes have also been designed, such as [23, 12, 7]. However, these schemes are mainly to support confidential communication among group members, whereas mSSL must handle not only confidentiality, but also other security functionalities such as integrity and proof of service.

7.2 Data Integrity Mechanisms

Existing peer-to-peer file-sharing applications provide data integrity functionalities. PROOFS [21] and Slurpie [20] recommend the use of MD5 or similar checksum algorithms. BitTorrent [5] adopts a superblock-based mechanism, which can have a high startup latency (as we discussed in Section 3). Solutions based on a Merkle hash tree have also been proposed for peer-to-peer environments, such as [6]. Different from those works, mSSL does not require pre-downloading of hash values, and for each block a client can just request an *integrity path* instead of a normally much longer *authentication path*. Integrity solutions also exist in different contexts; for example, TESLA [16] allows a large number of recipients to check the integrity of packets being delivered from a single source.

Researchers have also proposed solutions that are complementary to mSSL's integrity solution, including storage mechanisms of block-level integrity information (such as [15]), optimization of Merkle hash trees (such as [22]).

7.3 Offering Proofs of Service

Proof of service in this paper can be regarded as one particular case of a non-repudiation service. There have been quite a few non-repudiation schemes designed in different contexts, focusing on non-repudiation of origin, receipt, submission, and delivery [10, 13]. Verification of non-repudiation schemes have also been studied [26, 19, 11].

Proof of service is also similar to the strong fair exchange of information. In the context of this paper, fairness would mean for a provider to receive a proof of its service and for a recipient to receive the desired data. Solutions with a TTP can be created using an inline TTP (such as [4, 2] where the TTP is required to mediate every communication between a sender and a receiver), using an online TTP (such as [24, 25] where the sender and the receiver can directly communicate, but still need the TTP to store and fetch information), and using an offline TTP (such as [1, 9], where the TTP will be involved only when a problem occurs).

The most closely related to this paper is the fair exchange with an offline TTP. While leveraging current schemes, our solution for the hybrid peer-to-peer paradigm has an important difference in that a server itself can act as a TTP for its own provider and recipient clients. This is also an inherent advantage for enforcing fairness. Further note that the server is also the original source of the data that a provider offers to a recipient, bringing us another advantage in designing a solution in this hybrid paradigm. If needed, a server can verify the data without requesting them from other nodes, thus avoiding a drawback in many TTP-based solutions, especially when the amount of data is large.

8 Conclusions

As we continue to see the trend that the conventional client-server communication paradigm is enhanced with peer-to-peer communications among clients, often with dramatic advantages, serious security concerns arise in this hybrid communication environment. Compared to receiving data directly from a server, receiving data from arbitrary, often less trustworthy peer clients is subject to much higher security risks. Malicious clients may sneak into the system to corrupt the hybrid communications. Data integrity and data confidentiality are more easily breached. Mechanisms to reward peer clients for sharing data, such as crediting those providing data to others, are also vulnerable since clients may lie about peer-level service.

We designed and evaluated the mSSL protocol to address these security concerns. In contrast to the SSL protocol that protects conventional client-server communications, mSSL allows clients to share data from their server in a peer-to-peer fashion with a strong security. It protects both client-server and client-client communications. Furthermore, to strengthen the security of a wide range of Internet applications running with a hybrid communication paradigm, mSSL provides strong and flexible support for addressing both conventional security issues, such as client authentication, data integrity and data confidentiality, and new security issues such as proof of service in this special context.

mSSL's contributions also include its special attention to designing effective and efficient data integrity protection and proof-of-service mechanisms. It introduced an "integrity path" concept to allow prompt, block-level integrity verification with low communication and computation overhead. Its proof-of-service mechanism has minimal server overhead, uses small-sized proofs for the service of a very large number of blocks, and ensures the service of every block to be credited accurately. Our evaluation agrees with our design, and has shown an acceptable overhead under different scenarios where mSSL might be used.

References

[1] N. Asokan, V. Shoup, and M. Waidner. Asynchronous Protocols for Optimistic Fair Exchange. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 86–99, 1998.

[2] A. Bahreman and J. Tygar. Certified Electronic Mail. In *Proc. of Symposium on Network and Distributed Systems Security*, pages 3–19, 1994.

[3] BitTorrent, Inc. BitTorrent. `http://bittorrent.com`, 2005.

[4] T. Coffey and P. Saidha. Non-Repudiation with Mandatory Proof of Receipt. *SIGCOMM Comput. Commun. Rev.*, 26(1):6–17, 1996.

[5] B. Cohen. Incentives Build Robustness in BitTorrent. *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[6] A. Habib, D. Xu, M. Atallah, B. Bhargava, and J. Chuang. Verifying Data Integrity in Peer-to-Peer Media Streaming. In *Twelfth Annual Multimedia Computing and Networking (MMCN '05)*, 2005.

[7] Y. Kim, A. Perrig, and G. Tsudik. Tree-Based Group Key Agreement. *ACM Trans. Inf. Syst. Secur.*, 7(1):60–96, 2004.

[8] K. Kong and D. Ghosal. Mitigating Server-Side Congestion in the Internet through Pseudoserving. *IEEE/ACM Trans. Netw.*, 7(4):530–544, 1999.

[9] S. Kremer and O. Markowitch. Optimistic Non-Repudiable Information Exchange. In *Proceedings of the 21st Symposium on Information Theory in the Benelux*, pages 139–146, Wassenaar, The Netherlands, 2000.

[10] S. Kremer, O. Markowitch, and J. Zhou. An Intensive Survey of Fair Non-Repudiation Protocols. *Computer Communications*, 25(17), 2002.

[11] S. Kremer and J.-F. Raskin. A Game-Based Verification of Non-Repudiation and Fair Exchange Protocols. *Lecture Notes in Computer Science*, 2154:551+, 2001.

[12] X. S. Li, Y. R. Yang, M. G. Gouda, and S. S. Lam. Batch Rekeying for Secure Group Communications. In *Proceedings of the 10th International Conference on World Wide Web*, pages 525–534. ACM Press, 2001.

[13] P. Louridas. Some Guidelines for Non-Repudiation Protocols. *SIGCOMM Computer Communication Review*, 30(5):29–38, 2000.

[14] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. IETF RFC 2018: TCP Selective Acknowledgement Options, 1996.

[15] A. Oprea, M. Reiter, and K. Yang. Space-Efficient Block Storage Integrity. In *The 12th Annual Network and Distributed System Security Symposium*, 2005.

[16] A. Perrig, R. Canetti, J. D. Tygar, and D. Song. The tesla broadcast authentication protocol. 5(2):2–13, 2002.

[17] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley, 2000.

[18] R. Rivest and B. Lampson. SDSI: A Simple Distributed Security Infrastructure. http://theory.lcs.mit.edu/~cis/sdsi.html, 1996.

[19] S. Schneider. Formal Analysis of a Non-Repudiation Protocol. In *CSFW '98: Proceedings of the 11th IEEE Computer Security Foundations Workshop*, page 54, Washington, DC, USA, 1998.

[20] R. Sherwood, R. Braud, and B. Bhattacharjee. Slurpie: A Cooperative Bulk Data Transfer Protocol. *IEEE INFOCOM*, 2004.

[21] A. Stavrou, D. Rubenstein, and S. Sahu. A Lightweight, Robust P2P System to Handle Flash Crowds. In *the 10th ICNP*, pages 226–235, 2002.

[22] D. Williams and E. G. Sirer. Optimal Parameter Selection for Efficient Memory Integrity Verification Using Merkle Hash Trees. In *Proceedings of Network Computing and Applications, Trusted Network Computing Workshop*, 2004.

[23] C. K. Wong, M. Gouda, and S. S. Lam. Secure Group Communications Using Key Graphs. *IEEE/ACM Trans. Netw.*, 8(1):16–30, 2000.

[24] N. Zhang and Q. Shi. Achieving Non-Repudiation of Receipt. *The Computer Journal*, 39(10), 1996.

[25] J. Zhou and D. Gollmann. A Fair Non-Repudiation Protocol. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 55–61, Oakland, CA, 1996.

[26] J. Zhou and D. Gollmann. Towards Verification of Non-Repudiation Protocols. In *Proceedings of 1998 International Refinement Workshop and Formal Methods Pacific*, pages 370–380, Canberra, Australia, 1998.