# Generating Unforgeable Evidence for Secure Communications

Minwen Ji, Tom Rodeheffer[1], Marcos Aguilera, Mark Lillibridge
HP Laboratories Palo Alto
HPL-2005-71
April 21, 2005*

non-repudiation, evidence, secure socket layer, security protocol, digital witness

The Internet is increasingly being used for serious transactions involving money and goods. However, there hardly exists any means for users of Internet transactions to obtain third-party verifiable evidence of the transactions in which they participate. As a solution to this problem, we propose the use of a semi-trusted third party, called a digital witness, that can provide unforgeable transaction evidence without ever seeing the transaction contents. The witness is attached to secure communication sessions in an unintrusive manner: it requires no changes to existing servers or protocols. The key idea in enabling such a witness service is a novel algorithm that splits the computation of a message authentication code (MAC) between the client and the witness. We have implemented a prototype witness, tested it with real web sites and measured its performance. The experimental results show that the overhead of a witness is comparable to that of a web proxy. Therefore, the witness design can be implemented reasonably efficiently in practice.

* Internal Accession Date Only
[1]Microsoft Research
Approved for External Publication

# Generating Unforgeable Evidence for Secure Communications

Minwen Ji†, Tom Rodeheffer§, Marcos Aguilera†, and Mark Lillibridge†
†*Hewlett Packard Laboratories,* §*Microsoft Research*

## Abstract

The Internet is increasingly being used for serious transactions involving money and goods. However, there hardly exists any means for users of Internet transactions to obtain third-party verifiable evidence of the transactions in which they participate. As a solution to this problem, we propose the use of a semi-trusted third party, called a *digital witness*, that can provide unforgeable transaction evidence without ever seeing the transaction contents. The witness is attached to secure communication sessions in an unintrusive manner: it requires no changes to existing servers or protocols. The key idea in enabling such a witness service is a novel algorithm that splits the computation of a message authentication code (MAC) between the client and the witness. We have implemented a prototype witness, tested it with real web sites and measured its performance. The experimental results show that the overhead of a witness is comparable to that of a web proxy. Therefore, the witness design can be implemented reasonably efficiently in practice.

## 1 Introduction

The Internet is increasingly being used to conduct serious business, such as stock and bond trading, shopping, participating in auctions, obtaining quotes or financing, and paying bills. These activities all involve transactions of money, goods, or offers for the same. Yet, the Internet is a relatively dangerous venue for such dealings, so security is an important concern.

Basic security for these Internet transactions is often provided by the Secure Sockets Layer (SSL) protocol [7], a client-server-based protocol that guards against external parties attempting to eavesdrop on or tamper with the communication between the client and the server, and optionally against impersonation of the server or the client.

Besides attacks from external parties, users of Internet transactions may also suffer from misbehaviors of the parties they are conducting transactions with, *e.g.*, the web site owners. For example, a merchant might promise one price during a purchase transaction, but bill a higher price later. Or, an insurance company might "lose" an insurance claim submitted online and refuse to pay it when submitted again after the deadline. The authors know a real-life incident in which an individual lost a considerable sum of money when his stock broker failed to execute a trade within the promised window. These misbehaviors need not be due to malice: software or hardware failures can also result in lost or incorrectly performed transactions. In contrast to old-fashioned transactions, where the users receive the merchants, the services or at least hardcopies of the receipts immediately, users of Internet transactions are merely reminded to save or print the web pages that contain summaries of the transactions. Unfortunately, saved or printed web pages can hardly serve as convincing evidence of the transactions because they are easily alterable.

Clearly, a means for users to obtain third-party verifiable evidence of any Internet transactions in which they participate would help improve the security of the Internet as a venue for serious business. An evidence-generating scheme should meet the following requirements in order to be acceptable in practice:

- **Evidence integrity**: The resulting evidence should not be forgeable by the clients so that it is convincing to third-party reviewers.
- **Message confidentiality**: To preserve privacy, the contents of transactions should not be revealed to any parties other than the clients, the servers, and the intended reviewers.
- **Manageable changes**: Changes to existing systems and protocols should be minimal, and should be transparent to parties who do not receive a direct or substantial benefit from the changes.
- **Acceptable performance**: The evidence-generating processes should not significantly slow down the transactions.

There are two complementary approaches to generating evidence of Internet transactions, the server-initiated approach and the client-initiated approach. In the server-initiated approach, a server is willing to and is capable of issuing a third-party verifiable record of the transaction, such as a digitally signed [18] receipt. In the client-initiated approach, a client resorts to other means for obtaining evidence, with or without the server's awareness or cooperation. The latter approach is useful in situations where the server is unwilling to issue a digitally signed transaction record (*e.g.*, for fear of additional liability), or the server is incapable of doing so (*e.g.*, due to the lack of a Public Key Infrastructure), or the client wants the record for portions of the transaction that are not covered

in the server-issued receipt. In fact, such situations are quite common in current Internet experiences.

This paper proposes a client-initiated approach to generating unforgeable evidence, in which a *witness* is attached to a secure communication session in such a way that it can safely testify about the messages communicated in the session, without actually knowing the contents of the messages. It does not require any changes on the server side or in the communication protocol. This is important for very large systems with many owners of diversed interests, such as the Internet.

Once available, such a witness service would likely find uses other than protecting users from misbehaved merchants. For example, mortgage lenders might start accepting "witnessed" bank statements online as partial proof of a would-be borrower's finances; chess players might present verifiable logs of their online games for record setting purpose; and employers might record their employees' online transactions for training or quality assurance.

Two issues arise from the proposal of the witness service. The first is whether the witness scheme can meet all the technical requirements as discussed above. The second is whether there will be sufficient market demand to economically justify the deployment of such a witness service. The first issue can be addressed by a technical discussion on the design and implementation of the scheme, while the second issue is essensially a prediction of the profitability of a business running the witness service. In this paper, we concentrate on the technical aspects of the scheme, and leave the business aspects to the experts in the respective areas.

The technical contributions of this paper are the following. First, we design a novel algorithm that efficiently splits the computation of a Message Authentication Code (MAC) between two distrusting parties. Second, using this algorithm, we design and implement a witness service for SSL communication that provides evidence integrity, message confidentiality, manageable changes and low performance overhead. Third, we conduct a variety of experiments and demonstrate the robustness and efficiency of the implementation.

Since it is implemented at the SSL level, the witness service can work for any applications that run on top of SSL, such as HTTPS (which most secure web sites use) and secure FTP.

The rest of the paper was written with the assumption that the readers are familiar with the SSL protocol and the one-way hash functions it uses to construct MACs. Appendix A gives some background for readers who are less familiar with this material.

# 2 Use of existing techniques

In this section, we briefly discuss several approaches that use existing techniques to provide evidence for transactions conducted over the SSL protocol, and why they do not meet the requirements of evidence integrity and message confidentiality, as stated in Section 1.

**MACs** One might wonder if the client could simply use the MAC received from the server as the evidence that an SSL record was sent by the server, since the MAC could only be computed with the appropriate MAC secret. The problem, however, is that both the client and the server know the MAC secret, so that the client itself could have created a record and computed the MAC using the server's sending MAC secret. Therefore, this approach does not meet the requirement of evidence integrity.

**HTTPS proxy dump** An HTTPS proxy is a third party that is interposed between the client and the server and blindly forwards all traffic between them [21]. One might be tempted to have the proxy dump the traffic it forwards and use it as the evidence. In other words, the HTTPS proxy is used as a "witness" for the HTTPS communication. Such an approach does not compromise message confidentiality, because the proxy only sees encrypted data. However, it is the client, not the proxy, that authenticates the server during the SSL handshake protocol. Consequently, the client is able to mislead the proxy about the server's identity. Therefore, this approach does not meet the requirement of evidence integrity.

**Fully trusted witness** Another obvious way of building a witness is to have the witness establish two independent SSL sessions, one between the server and the witness, and the other between the witness and the client. The witness decrypts the messages from one session, encrypts them using the keys from the other session, and forwards the re-encrypted messages to the other session. This lets the witness authenticate the server and every message. However, since the witness sees the plaintext of the messages (*e.g.*, account numbers and passwords), the user has no privacy. Therefore, this approach achieves evidence integrity, but not message confidentiality.

# 3 Design

In this section, we describe our design of a witness for the SSL protocol. We start with a system model in which the various parties involved in a witnessed SSL session and the assumptions on their trust relationships are explained. We then show how to achieve the security goals of the witness using a novel algorithm called MAC decomposition.

## 3.1 System model

In order to generate unforgeable evidence of a transaction conducted over an SSL session, we attach a witness to the SSL session. The server operates in the same way as in a regular SSL session. The client records its communication


Figure 1: Witnessed SSL communication.

with the server in plaintext, while the witness records "evidence" of the communication, as will be explained later. The evidence is signed by the witness in the end of the session. Together with the plaintext, it can be used as irrefutable proof that the communication took place as recorded. The whole scheme is illustrated in Figure 1.

We call the communication record produced by the client *verified* communication record *as attested to by* witness $W$ if it is consistent with the evidence that was signed by $W$. We call a party that performs this consistency check a *verifier*.

The trust relationships in the system are as follows:

- The witness and the verifier trust neither the server nor the client.
- The verifier trusts the witness to produce honest testimony. This is analogous to the relationship between a reviewer of a notarized document and the licensed public notary.
- The client trusts that the witness is not both malicious and able to intercept messages sent directly between the client and the server. We call this a *semi-trusted* relationship.
- The server is not aware of the presence of the witness during the communication with the client.

A verifier that trusts witness $W$ is able to infer a reasonable amount of information about a witnessed SSL session from the verified communication record attested to by $W$, including the server's identify and the messages transmitted. In particular, the witness guards against situations where the server denies sending a message that it actually did during the witnessed period, and situations where the server falsely claims receiving a message that the client did not actually send during the witnessed period.

## 3.2 Design constraints

In order to meet the requirements for evidence-generating schemes as discussed in Section 1, we use the following constraints to guide our witness design:

- In the beginning of each SSL session, the witness, not the client, should authenticate the server and generate the shared secrets with the server. This prevents the client from colluding with a server impostor or forging the evidence.
- The addition of a witness to an SSL session should not enable any parties, including the witness itself, to eavesdrop, tamper or forge the messages between the client and the server. This ensures the same level of protection for the client as the regular SSL protocol does.
- No changes should be required on the server side or in the SSL protocol. This leaves the client the only adopter of the changes, which is reasonable because the client is the main beneficiary of the witness service.
- The performance overhead of a witness should be comparable to that of existing services on the Internet, such as a web proxy. This makes the witness overhead more likely to be acceptable.

## 3.3 Decomposed MAC

The main challenge in designing the witness is to prevent the client from tricking the witness into testifying about messages that have not been exchanged between the client and the server, while preserving the client's privacy from the witness. Our key idea to address this challenge is to split the computation of the MAC in SSL between the client and the witness in such a way that produces the same results as the regular MAC function, without giving the MAC secret to the client or the message content to the witness. We call this method of computing MAC *decomposed MAC*.

We made an important observation that the MAC function in SSL, in particular the HMAC function [3], can be computed in three steps such that each step requires only the message content or only the MAC secret, but not both. This is attributed to the *iterated* nature of the underlying hash functions in HMAC, *e.g.*, MD5 [17] and SHA1 [12], which start with a constant Initialization Vector (IV), process an arbitrary-length message in blocks of fixed size and use the result of the current block as the IV for the next block. The padding in HMAC, which was originally designed to optimize performance, also plays an important role in decomposing HMAC. The MAC of an SSL 3.0 record is computed by

$$\begin{aligned}&\mathrm{HMAC}(sec, seq, l, content) = \\ &\quad H(sec + pad_2 + InnerHash),\end{aligned}$$

and

$$\begin{aligned}&InnerHash = \\ &\quad H(sec + pad_1 + seq + l + content),\end{aligned}$$

where $H$ is an iterated one-way hash function, $sec$ is the MAC secret, $seq$ is the sequence number of the SSL record, $l$ is the length of the record, $content$ is the record content, $pad_1$ is the byte $0x36$ repeated 48 times, and $pad_2$ is the byte $0x5c$ repeated 48 times [13]. Definitions of the HMAC terms can be found in Appendix A.2.

Let $sec + pad_1'$ be the first block of the message hashed for $InnerHash$ where $pad_1 = pad_1' + pad_1''$. ($pad_1''$ is zero bytes for MD5 and four bytes for SHA1.) In the first step of the decomposed MAC computation, the witness computes and then sends to the client

$$DeHMAC_1 = H(sec + pad_1')$$

In the second step, the client computes and then sends to the witness

$$DeHMAC_2 = H_{DeHMAC_1}(pad_1'' + seq + l + content)$$

where $H_{DeHMAC_1}$ denotes the hash function $H$ with $DeHMAC_1$ as its IV. Because of the iterated nature of $H$ and the fact that $sec + pad_1'$ exactly fills a block, $DeHMAC_2$ is equal to $InnerHash$. In the third step, the witness computes

$$DeHMAC_3 = H(sec + pad_2 + DeHMAC_2)$$

which is equal to the normal HMAC.

In a witnessed SSL session, when the client wants to send an SSL record, it proceeds as if it is in a regular SSL session except that instead of computing the regular MAC locally, it computes the decomposed MAC jointly with the witness. The witness records the computed MAC as evidence. Receiving a record is similar except that the witness does not give the computed MAC to the client, but only tells it whether the received MAC is the same as the computed MAC.

In the decomposed MAC computation, the client gets a one-way hash value of the MAC secret ($DeHMAC_1$), and the witness gets a one-way hash value of the message content ($DeHMAC_2$). By definition, a one-way hash function $H$ has the following property: given a hash value $h$, it is cryptographically hard to compute a message $m$ such that $H(m) = h$. It follows that it is cryptographically hard for the client to compute the secret from $DeHMAC_1$ and for the witness to compute the message content from $DeHMAC_2$. Therefore, the decomposed MAC is safe against the client's forgery while it preserves message confidentiality from the witness.

As long as the communication channel between the client and the witness is secure, an external party cannot gain any more information about the witnessed SSL session than it could have gained about a regular SSL session. Therefore, the witnessed SSL protocol provides the same protection from an external party as the regular SSL protocol does.

## 3.4 SSL connection handover

In the beginning of a regular SSL session, the client invokes the *SSL Handshake Protocol* to authenticate the server's identity and to generate shared session keys and secrets with the server. (Definitions of the SSL terms can be found in Appendix A.1.) In a witnessed SSL session, sssuming that the server, the client, and the witness are running on three independent sites, the witnessed SSL handshake consists of the following steps:

1. The client establishes two TCP connections, one with a server and the other with the witness, and then informs the witness that it wants to start an SSL connection.
2. The witness performs an SSL handshake with the server to establish a new SSL connection, recording the results, including the MAC secrets, the encryption keys, the verified server certificate and the selected cipher suite.
3. The witness computes the first step of the decomposed MAC ($DeHMAC_1$) and copies the *partial* state of the SSL connection (including $DeHMAC_1$ and the encryption keys, but not the MAC secrets) to the client through a secure channel. We call this operation a *handover*.

Once the handover is done, the client can start exchanging application data with the server. A web user often starts an *application session* with an HTTPS server by logging into her personal account on the server with her password. It is worth noting that the SSL session that the witness establishes with the server is independent of the application session, and does not require the witness to know the client's application-level secrets, such as passwords. Rather, the application session is layered on top of the SSL session.

The witness's participation in the handshake with the server allows the witness to directly authenticate the server, preventing the client from impersonating the server. For example, after the handshake the witness is convinced that it is communicating with the server who owns the private key corresponding to the public key in the verified and recorded certificate.

The witness copies the encryption keys to the client after the handshake so that the client can have confidential communication directly with the server. The witness keeps the MAC secrets from the client so that the client cannot produce valid messages without the witness's help. The encryption keys and MAC secrets in SSL are independent of each other, albeit derived from the same shared secret. Therefore, it is impossible for the client to guess the MAC secrets even when given the encryption keys.

## 3.5 Client proxy

Figure 2 shows the layering and end points of connections in a witnessed SSL session. The SSL session is established between the server and the witness, and partially shared with the client. The application session resides between the client and the server, in the same way as in a system without the witness. TCP connections exist only between the client and the server and between the client and the witness. That is, the

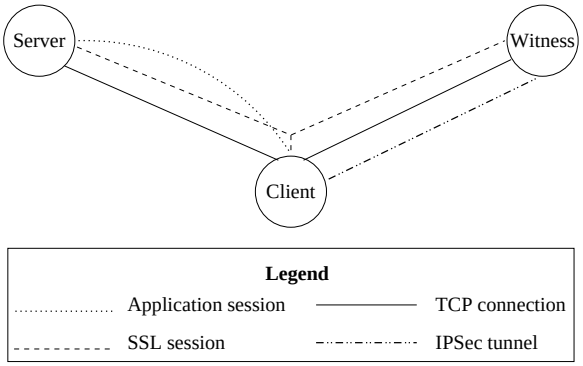


Figure 2: Connection hierarchy in a witnessed SSL session. The IPSec tunnel in this figure is an example of how the channel between the client and the witness can be secured.

witness communicates with the server through the client as a *proxy*. There are two reasons for this design decision.

First, many encryption engines maintain internal state that is dependent on the stream of bytes that have been processed in the past. Therefore, in order to keep the state of the encryption engine in the client consistent with that in the SSL server, it is easiest to have all messages encrypted or decrypted solely by the client once the handover of the SSL connection is done. Before the handover is done, the witness encrypts or decrypts all messages. An alternative is to process the messages in part by the client and in part by the witness, and to explicitly synchronize the encryption engine state between them. This alternative is unattractive because it needs to be implemented specifically for every type of encryption engine that the SSL protocol supports.

Second, since the witness knows the SSL encryption keys and MAC secrets, it is able to eavesdrop on or forge the messages exchanged between the client and server if it can intercept those messages or impersonate the client or server. To make these attacks harder to launch by the witness, we have the client exchange application data with the server directly, bypassing the witness. We assume that the witness is not on the network path between the client and the server,[1] and hence cannot simply sniff the traffic between them. We discuss this in more detail in Section 3.9.

All messages between the client and the witness, including the witness-server communication through the client proxy, are transmitted over a secure channel that is separate from the SSL session with the server; *i.e.*, one with independent authentication. This secure channel can be implemented by Secure Remote Procedure Calls, by an independent SSL session, or by a TCP connection on top of a secure IP tunnel (*e.g.*, an IPSec tunnel).

[1]This prohibits Internet Service Providers from offering a witnessing service to their clients.

## 3.6 Message exchanges

In a witnessed SSL session, all SSL records of application data are sent or received by the client, not the witness. In HTTPS, the application data includes URL requests, HTML pages, cookies, etc. Before sending an SSL record to the server, the client computes its MAC jointly with the witness and then encrypts the record. After receiving an SSL record from the server, the client decrypts the record and sends its MAC to the witness, who then verifies it via the decomposed MAC procedure. If the verification fails, the witness shuts down the connection and invalidates the session. The client records the connection numbers, sequence numbers, lengths, and unencrypted contents of all the application data records it has sent or received. The witness records all the MACs it is asked to compute or verify, together with the corresponding connection numbers, sequence numbers (determined independently of the client), directions (from client or server), and the current time.

All SSL control messages, including handshake messages, change-cipher-spec messages and alert messages, are initiated or handled by the witness, although physically transmitted through the client proxy. After sending or receiving a control message, the witness informs the client of any necessary state updates. The witness uses the regular MAC function to compute MACs for control messages because decomposed MAC is unnecessary for control messages and is slower.

Detailed procedures for sending an SSL record from the client or the witness to the server can be found in Figure 8 in Appendix B. Detailed procedures for receiving an SSL record from the server can be found in Figure 9 in Appendix B. Receiving an SSL record is considerably more complicated than sending a record because it needs to handle unexpected and out-of-band messages (*e.g.*, alerts or session renegotiation requests) from the server.

## 3.7 Verification of records

When it needs to prove to a verifier the contents of a witnessed transaction, the client asks the witness for signed evidence of the SSL communication involved. The witness will then digitally sign and send to the verifier the recorded server certificate, the cipher suite chosen, the per-connection MAC secrets, and ordered lists of per-message information. The per-message information consists of the SSL-connection number, the direction the message was transmitted, the message's sequence number, the message's MAC, and the time the witness computed that message's MAC. Messages are listed in the order their MACs were computed by the witness.

Signed evidence is provided only for successfully completed SSL sessions. While we could provide some evidence for invalidated sessions (due, for example, to a fatal alert received from the server or an incoming message's MAC failing to verify), the obtainable evidence is hard to interpret

and there seems to be no compelling reason to handle these cases. Note that if the server attempts to take advantage of this limitation by abnormally terminating sessions once a user's application-level transaction has completed, the user's browser will complain, alerting them. Because a server has no way of telling which clients are using a witness, doing this means that everyone using the site will receive SSL errors, risking alienating a large number of customers.

If the server, the client, or the witness crashes during an SSL session, the witnessing service will be aborted and no evidence will be provided about the session.

The verifier first authenticates the signed evidence using the public signature key of the witness. Provided the evidence has the correct signature, the verifier then compares it to the communication records provided by the client. The verifier does this by lining up the list of communication records next to the list of per-message information in the evidence and then comparing side-by-side entries. (Verification fails if the lists differ in length.) A pair of entries compare successfully iff they have the same connection number, sequence number, direction, and MAC, where the communication record's MAC is obtained by computing the regular MAC function on it using the appropriate MAC secret and cipher suite from the evidence. If all entries compare successfully, then the verifier concludes that the records are consistent with the evidence and marks them as verified.

## 3.8 Analysis of records

Given a set of verified communication records attested to by a witness that a verifier trusts, what conclusions can that verifier safely make about those verified communication records?

Since the client did not know any of the server's sending MAC secrets on an SSL connection while the connection was being witnessed and is unable to alter the evidence afterwards due to the witness's signature, every MAC recorded as received from the server must have been generated by the authentic server. Therefore, the corresponding messages in the client's records must have been sent by the server rather than forged by the client. Similarly, since the client did not know any of the witness's sending MAC secrets, the server could not have received any messages from the client while the communication was being witnessed that are not recorded as sent by the client. Because each direction of each connection uses different MAC secrets and the MAC covers the sequence number, the verifier can be sure that the information associated with each record is correct.

When could the client have omitted telling the witness about a received message? Because the sequence numbers of the recorded records have no gaps (otherwise verification would have failed), the only received messages the client could have omitted were ones received after the recorded received messages. Thus, the messages recorded as received on a connection are a prefix of the messages actually received by the client. Furthermore, if a close-notify message from the server was recorded at the end of a connection, then the recorded messages are an exact match of the received messages. This is because the client would not have been allowed to accept more messages from the server after it received a close-notify message.

When could the client have failed to send a message that it claimed it sent? Once a client drops an outgoing message, no further valid messages can be sent on that connection while that connection is being witnessed, due to SSL's sequence number checking. The client cannot reuse or skip sequence numbers because the sequence numbers it records are compared to the witness' independently-assigned sequence numbers. Therefore, the verifier can conclude that the messages actually sent by the client on a connection while that connection was being witnessed are a prefix of the messages recorded as sent on that connection.

In addition, since the evidence includes a time for each recorded MAC, the verifier can conclude that a message recorded as received by the client at time $t$ cannot have been sent by the server after that time and that a message recorded as sent to the server at time $t$ cannot have been received by the server before that time.

The witness's evidence is provided only for the period in which the witness is used. The evidence is not intended to cover the client-server communication before the witnessed SSL session is established or after the evidence is revealed to a verifier or to the client itself, because the witness has no control of such communication.

Since the SSL protocol keeps separate sending and receiving sequence numbers, the verifier is unable to determine the exact temporal interleaving between the sent and received sequences. However, the verifier can be sure that the messages recorded as sent by the server could be replying only to the messages recorded as sent by the client. If necessary, the temporal interleaving can be determined with application-specific knowledge. For example, in the HTTPS protocol, once the SSL records are assembled into HTTP requests and replies, the interleaving of those messages becomes obvious.

## 3.9 Client Privacy

The client's privacy depends on the witness being either unwilling or unable to intercept the client's direct communication with the server. While not ideal—ideally, the client would not have to place even this much trust in the witness—this is not without precedence; Abadi, *et al.* [1], for example, provide a similar level of privacy from a semi-trusted certified e-mail provider.

The only way we know of to remove the need for the client to (fully) trust the witness while still leaving the server unchanged involves the use of secure multi-party computation (SMPC) [22] to allow the client and witness to conduct a joint handshake with the server such that only the witness

gets the resulting MAC secrets and only the client gets the resulting encryption keys. Very briefly, this would work by having the client and witness each submit a random number to the SMPC, which would be summed to generate the shared secret, which in turn would be encrypted with the server's public-key and used to generate the MAC secrets and encryption keys. Unfortunately, all the SMPC algorithms we know of have unacceptable performance for computations of this size.

Why not assume a *fully trusted* witness who is trusted with the full content of the client-server communication and use the far simpler technique discussed in Section 2? We contend that a semi-trusted witness is more desirable than a fully trusted one, for several reasons:

First, there is no legal way for a semi-trusted witness to access the communication content, while it is legal for a fully-trusted witness to do so. Second, a semi-trusted witness, even when it is corrupt, may be incapable of intercepting the client-server communication because the targeted packets are not routed near the witness on the Internet. There are known attacks for intercepting Internet traffic, such as DNS spoofing [20] and TCP hijacking [11], but they are becoming less effective as the Internet improves and they are usually detectable quickly. Once a witness is caught using such attacks, it would quickly lose its reputation. In contrast, a corrupt fully-trusted witness can eavesdrop or tamper with the communication content with entirely internal operations, and hence remain undetected for a much longer time.

Using a semi-trusted service is analogous to sending a sealed letter rather than a postcard via the post office. A postcard corresponds to full trust since its contents are available immediately to the postal service. A sealed letter, by contrast, corresponds to semi-trust, because it provides a reasonable amount of privacy while not being absolute: specialists can (sometimes undetectably) open letters with ingenious apparatuses. Doing so, however, is prohibited by legislation in most countries.

# 4 Implementation

We have implemented a prototype of the witness scheme on the Linux platform; the prototype consists of a witness, a witness-aware SSL library, and a program for verifying communication records. The witness-aware SSL library is a modified version of the open-source SSL library, OpenSSL 0.9.7, and is used by both the witness and clients. We enable client software to work with the witness by linking it to this modified SSL library, either statically or dynamically. No other modification to existing client software is necessary if the client software already uses OpenSSL. If the client software was dynamically linked to the OpenSSL library, not even recompilation is necessary.

The witness is implemented as a Remote Procedure Call (RPC) server. It handles two types of client requests: SSL connection establishment and shutdown (*e.g.*, when a client calls SSL_connect() or SSL_shutdown()), and decomposed MAC computation for application data records. A single witness RPC server can handle multiple clients and multiple SSL sessions/connections per client.

The client proxy is implemented as a thread in the same address space as the client application. Each OpenSSL application first creates a structure called *SSL context*, which it then uses to make SSL connections in the created context. The context consists of information such as a preferred cipher list and session cache. When a new context is created for a witnessed SSL client, the witness-aware library creates an RPC handle to the witness and forks a proxy thread for that context. The proxy accepts TCP connections from the witness by listening on a port; this port number is sent to the witness in the first call to the witness RPC server within this context. There is a witness-proxy connection for each client-server SSL connection. To reduce the overhead for three-way TCP connection establishment, we keep a witness-proxy connection in a pool of "free" connections after its corresponding SSL connection is shut down, and reuse the freed connection when a new client-server SSL connection is established.

The client proxy repeatedly polls the per-connection server sockets and witness sockets, the free sockets, and the per-context RPC socket. For each control message that arrives at a per-connection server/witness socket, the proxy decrypts/encrypts it as necessary and forwards it to the corresponding witness/server socket. Each message that arrives at a free socket indicates the need for a new witness-proxy connection and hence converts the free socket into a per-connection witness socket.

We have implemented two alternatives to handle application data in the client proxy. The first one is a naive, synchronous method. For each application data record that arrives at a per-connection server socket, the proxy forwards it to the application, which then makes a synchronous RPC to the witness for MAC verification. In the synchronous method, the MAC computation for the next record will not be started until after the processing of the current record is completed. Therefore, the communication overhead to the witness for sending or receiving $n$ records is $n$ times the round trip latency between the client and the witness.

The second method is an optimized, asynchronous method. It reduces the witness overhead for sending or receiving large amounts of application data by pipelining the MAC computation on multiple records and by overlapping communication to the witness with communication to the server. For each record that arrives at a server socket, the proxy adds the record to a *pending records queue*, makes an asynchronous RPC to the witness for MAC verification, and continues to process the next message without waiting for the reply of the current RPC. When the reply for a MAC request arrives at the RPC socket, the proxy stores the reply with the relevant

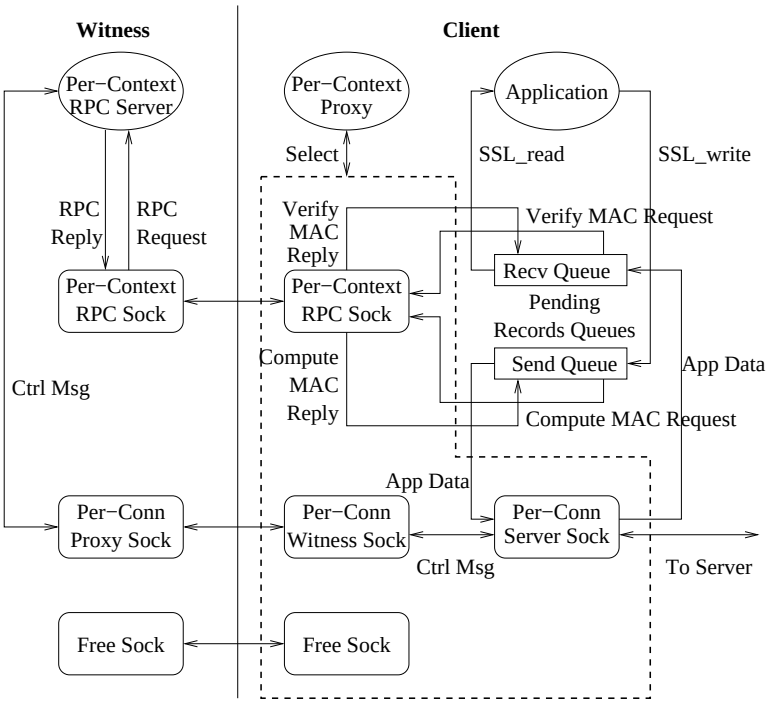


Figure 3: Components and messages in the witness prototype.

pending record in the queue and marks the record as ready for the application. When it tries to read an SSL record by calling SSL_read(), the application retrieves the first ready record with its MAC result and removes it from the queue. The use of the pending records queues enables parallelism in processing and transmitting multiple SSL records.

Figure 3 shows the data and control flows between the components in the witness prototype.

During recording, the witness-aware SSL library writes each MAC (for the witness) or SSL record (for the client) to a separate file with the connection identifier (unique within an SSL session), direction (sending or receiving) and sequence number in the file name.

The prototype implementation involves roughly 7,000 lines of manually written C code and 1,000 lines of automatically generated RPC stubs.

# 5 Robustness tests

Links [15] (different from Lynx) is an open-source, text-based Web browser that is included by default in some RedHat Linux installations. It supports HTTPS through the dynamically-linked OpenSSL library (`libssl.so`). We replaced the system's default OpenSSL library with the witness-aware library and ran the witness on one of the local machines. We then invoked the unmodified browser to access real-world HTTPS sites that the authors have personal accounts on, including online merchants (*e.g.*, `amazon.com`), phone companies (*e.g.*, `verizonwireless.com`), calling card companies, etc. Each test started with logging onto a personal account and continued with browsing web pages or making test transactions, such as changing the account password and address.

A limitation of Links is that it does not support Javascript and hence cannot handle web pages that contain Javascript code. To further test the witness prototype, we modified the Mozilla browser source code so that it calls the witness-aware OpenSSL library instead of its own security library (*i.e.*, the Network Security Services (NSS) package). With the modified Mozilla browser, we were able to successfully use web sites that require Javascript support, including credit card companies (*e.g.*, `americanexpress.com`) and banks (*e.g.*, `addisonavenue.com`).

After a browsing session finished, we used a verifier program to successfully compare the contents recorded by the browser to the MACs recorded by the witness, following the procedure described in Section 3.7.

These tests helped us improve the robustness of the implementation, particularly the handling of non-blocking sockets, fragmented messages, connection shutdown, and callback functions. After these improvements, the witness-aware browsers were able to access any HTTPS site that the original browsers were able to access in the experiments.

| | |
|---|---|
| $L_s$ | Round-trip network latency between the client and the server |
| $L_w$ | Round-trip network latency between the client and witness |
| $L_d$ | Average disk seek and rotation latency on the client and witness machines |
| $B_s$ | Network bandwidth between the client and the server |
| $B_d$ | Disk write bandwidth on the client and witness machines |

Table 1: Variables used in the analysis of witness overhead.

# 6 Performance

An important aspect of the witness design and implementation is understanding the overhead that witnessing adds to regular SSL communication, and optimizing it if needed. In this section, we first analyze the witness overhead for an SSL handshake and for application data, and then report experimental results on the prototype.

## 6.1 Overhead analysis

Table 1 defines the variables used in the analysis. For simplicity, we assume that computation speed and disk access time on the client and the witness machines are the same.

### 6.1.1 Handshake messages

In order to initiate a handshake with the server, the client needs to make an RPC to the witness. In handling this RPC, the witness exchanges a number of messages with the server through the client proxy. All other computation and communication done in a regular SSL client, such as certificate verification and key generation, will be done in the witness instead. Let $C_h$ be the amount of time needed for such computation and communication. Since the messages during a handshake are small and are not recorded to disk, only network latencies matter in this case.

Therefore, the time needed for a regular handshake is $nL_s + C_h$, while the time needed for a witnessed handshake is $L_w + n(L_w + L_s) + C_h$, where $n$ is the number of round trips between the witness/client and the server during the handshake. There are 12 one-way messages for a full handshake in the SSL 3.0 specification, but less are needed in practice (*e.g.*, 9 in the experiments) because some of the 12 messages are optional. However, the messages are not exact request/reply pairs, but instead come in groups of multiple messages sent and multiple messages received. Assuming that the underlying network protocols (*e.g.*, TCP/IP) pipeline messages transmitted in the same direction, the number $n$ of round trips is actually 2 during a full handshake and 1 during a partial handshake. The additional round trip between the witness and the client is for the RPC request and reply. The overall witnessing overhead for a handshake is thus $(n+1)L_w$.

We compare the witness overhead to that of a regular, non-caching HTTPS proxy, which is interposed between the client and server and blindly forwards all traffic between them [21]. The HTTPS proxy overhead for a round-trip message exchange between the client and an HTTPS server is approximately the round-trip latency between the client and the proxy. Therefore, the proxy overhead for an SSL handshake is $nL_w$, which is comparable to the witness overhead. Since web proxies are widely deployed on the Internet today, such overhead appears to be acceptable to Internet users.

### 6.1.2 Application data

Under the synchronous method (Section 4), in order to send or receive an application data record, the client needs to make an RPC to the witness for a decomposed MAC computation. Since the RPC request and reply contain mainly *DeHMAC* values or the MAC verification result, all of which are small, only the network latency between the client and the witness matters. The witness needs to write the MAC and a small amount of other data to disk; here the disk seek and rotation time dominates. The client also needs to write the record's contents to disk. On both the client and the witness, at least two disk accesses are needed by the prototype, one for updating the metadata for a newly created file, and the other for writing the MAC data or record contents to the file. The total amount $C_a$ of computation needed for processing an application record (including MAC computation and encryption) is the same in both regular and witnessed SSL. Therefore, the time for sending or receiving a record of size $s$ is $s/B_s + C_a$ for regular SSL, and $s/B_s + C_a + L_w + 4L_d + s/B_d$ for witnessed SSL. The overhead per record is thus $L_w + 4L_d + s/B_d$.

Under the asynchronous method, the MAC computation on multiple records is pipelined. Therefore, the witness overhead for sending or receiving $r$ records of size $s$ each is the pipeline setup time plus $r-1$ times the pipeline bottleneck. The pipeline setup time is the witness overhead for the first record in the pipeline, $L_w + 4L_d + s/B_d$, and the pipeline bottleneck is the disk write time at the client, $2L_d + s/B_d$. Therefore, the average overhead per record, processed in batches of $r$ records, is

$$2L_d + s/B_d + \frac{L_w + s/B_d}{r}$$

We compare the witness overhead for batched SSL records in the asynchronous method to that of a regular, non-caching HTTPS proxy [21]. Based on the same pipeline time calculation as above, the HTTPS proxy overhead for sending or re-

ceiving a record of size $s$ in $r$-size batches is $s/B_p + L_p/(2r)$, where $L_p$ is the round-trip latency and $B_p$ is the network bandwidth between the client and the proxy. Assuming that $B_p$ is comparable to $B_d$ and that $s/B_d$ is negligible compared to $L_w$, the difference between the witness overhead and the proxy overhead is $2L_d + (L_w - L_p/2)/r$. If the disk seek and rotation latencies for batched records can be reduced by file system or disk scheduling optimization (such as NVRAM caching or request sorting and coalescing), then the witness overhead for batched records is comparable to that of a regular HTTPS proxy that has a round-trip latency of $L_p = 2L_w$ to the client.

#### 6.1.3 Scalability

The amount of computation and communication required on the witness is similar to that of a regular SSL server with little or no application-specific processing. Therefore, we expect that the witness will exhibit the same scalability (*i.e.*, supports the same number of clients) as a regular SSL server does.

### 6.2 Experimental setup

We ran experiments on the prototype to measure the overhead that witnessing adds to the response time of client SSL operations.

We conducted the experiments on an emulated wide area network (WAN) environment. 4 Celeron 400 MHz PC's were used. Machine 1 ran a popular WAN emulator on FreeBSD called Dummynet [19] in its bridging mode, and physically connects the other machines. Machine 2 ran an Apache web server with HTTPS support. Machine 3 ran the witness. Machine 4 ran a client test program that accessed the HTTPS pages on the Apache server, either directly or with the help of the witness. The web server, the client, and the witness communicate with each other through the WAN emulator.

We used Dummynet to control the delay of Ethernet packets transmitted between the other machines. We configured four pipes, one in each direction between the client and the server and one in each direction between the client and the witness. Each pipe was configured to hold 50 packets (1500 bytes each) and to delay half of the desired round-trip latency between the relevant machines. For example, to establish a round-trip latency of 100 ms, the delay for each pipe would be set to 50 ms. Since the effective bandwidth of a pipe is the product of the pipe capacity and the one-way latency, in this case the bandwidth is roughly 1.5 MB/s. By using Dummynet, the client, server, and witness applications observe network latencies similar to those in a WAN.

The client and the witness communicate over an IPSec tunnel running on top of the emulated WAN. The tunnel is provided by the IPSec implementation package FreeS/Wan [6]. The purpose of using IPSec is to provide a secure channel between the client and the witness while leveraging the RPC service available on Linux. Since IPSec protects data at the IP layer, it works transparently underneath RPC or any other IP application. In the experiments, IPSec adds about $400 + 0.4b$ microseconds to the one-way latency of each $b$-byte IP packet.

We toggled witness recording in the experiments. When the recording was turned on, each file was flushed to the disk synchronously. The client and the witness each have an IDE disk with a maximum bandwidth of approximately 25 MB/s and an average seek and rotation latency of 10 ms.

All SSL sessions created in these experiments used the cipher suite RSA-RC4-MD5. That is, RSA was used in the handshake for key generation, RC4 was used as the symmetric encryption algorithm, and MD5 was used as the underlying hash function in the HMAC function.

In the beginning of the each experiment, the client program measured the round-trip network latencies to the witness and to the server by pinging those two machines 10 times and calculating the average response times. The client then repeats the following procedure 100 times in each experiment: establishing an SSL connection with the HTTPS server (*i.e.*, socket connect and SSL_connect()), sending a CGI request to the server (*i.e.*, SSL_write()), receiving the CGI response from the server (*i.e.*, SSL_read()), and shutting down the SSL connection (*i.e.*, SSL_shutdown() and socket shutdown). We measured the elapsed time on the client machine for each of the above operations.

There are two types of CGI requests in the experiments, read intensive and write intensive. The read-intensive request carries an integer argument, $n_r$, which is randomly selected from the range 0–100K for each request. The CGI program on the server responds to the client with $n_r$ random bytes. We have observed that the Apache web server fragments the reply into SSL records which are typically 4 KB and never more than 4 KB in size. The write-intensive request posts 160 KB of random data to the server, using SSL records of size $n_w$, where $n_w$ is randomly selected from the range 512 bytes to 16 KB for each request. 16 KB is the maximum SSL record size. The CGI program on the server responds with a small fixed-size message.

The figures in Section 6.3 show the results of the 100 runs with randomized parameters in each experiment, averaged into bins for better readability.

### 6.3 Experimental results

Figure 4 shows the time for SSL handshakes (*i.e.*, SSL_connect()) as a function of the round-trip latency to the witness, called the *witness latency* hereafter. Every additional second in the witness latency increases the full handshake time by 5 seconds and increases the partial handshake time by 4 seconds. This is larger than the estimated overhead ratio, *i.e.*, 3 for a full handshake and 2 for a partial handshake (Section 6.1.1). Tcpdump output explains the difference. The witness RPC reply message was large enough to be fragmented

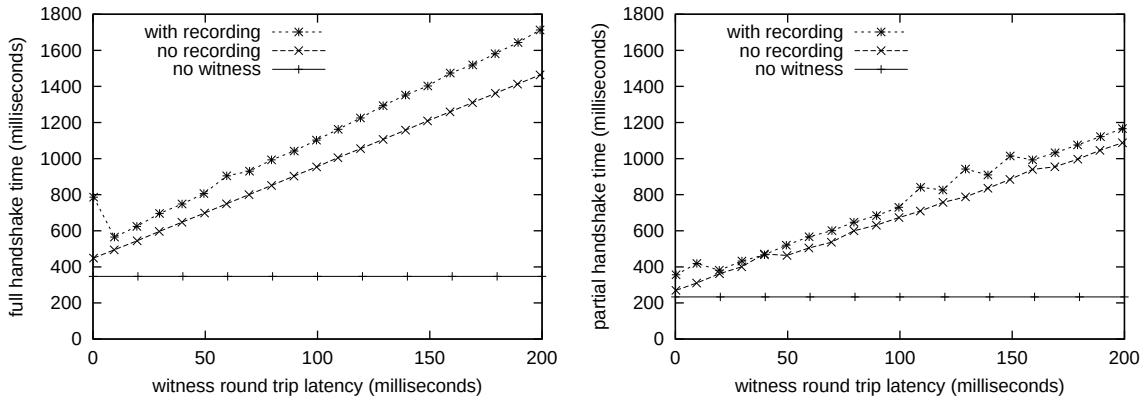


Figure 4: Handshake time vs. witness latency. The round-trip latency to the server is 100 ms in these experiments. The slope of the no-recording and with-recording lines is roughly 5 for full handshakes and 4 for partial handshakes.

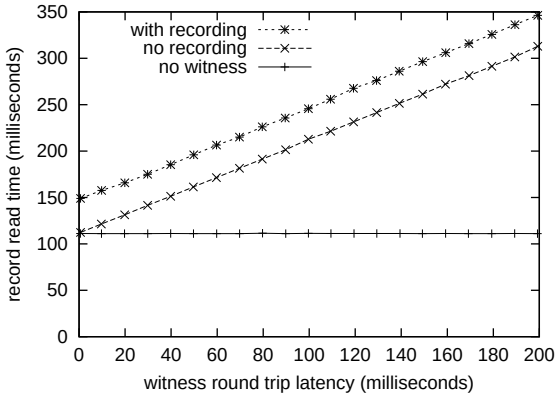


Figure 5: Record read time vs. witness latency. The round-trip latency to the server is 100 ms and the SSL record size is 4 KB in these experiments. The slope of the no-recording and with-recording lines is roughly 1.

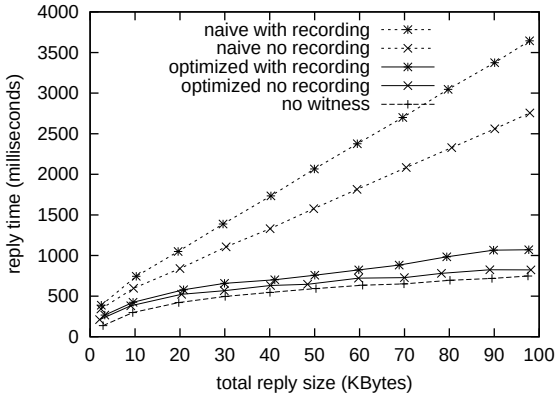


Figure 6: Reply time vs. reply size. The round-trip latency to the server and to the witness are both 100 ms, and the SSL record size is 4 KB or less in these experiments. The naive-no-recording line has a slope of 100 ms per 4 KB, and the naive-with-recording line has a slope of 131 ms per 4 KB. The average difference between the optimized-no-recording line and the no-witness line is roughly 80 ms. The difference between the optimized-with-recording line and the optimized-no-recording line ranges from 48 ms to 245 ms.

into 6 packets, and the TCP sender underlying the RPC channel at the witness waited for the acknowledgment of the first 2 packets before it sent the other 4, because it was in a slow start mode. Consequently, the RPC reply took 1.5 times the round-trip time instead of the estimated 0.5 times. TCP slow start also broke the pipeline for the transmission of the first few handshake messages between the client and the witness.

There is little difference between the witness lines with recording turned on and off, because only a minimal amount of information (*e.g.*, MAC secrets) is recorded during the handshake.

Figure 5 shows the SSL_read() time for a 4 KB SSL record as a function of the witness latency. The witness lines have a slope of roughly 1. In Section 6.1.2, we estimate that the witness overhead for each SSL record is the witness latency plus disk write time. In the experiments with the witness and no recording, the witness overhead is nearly equal to the witness latency. In the experiments with recording, the estimated disk write time is $4 \times 10\text{ ms} + 4\text{ KB}/(25\text{ MB/s}) \approx 40\text{ ms}$, which roughly matches the difference between the no-recording and with-recording lines.

Figure 6 shows the time for receiving a large reply (*i.e.*, multiple calls to SSL_read()) as a function of the reply size. Since the Apache Web server sends the reply by SSL records of 4 KB or less in size, larger reply sizes result in larger numbers of SSL records. Therefore, Figure 6 actually shows the time for receiving a batch of fixed-size SSL records as a

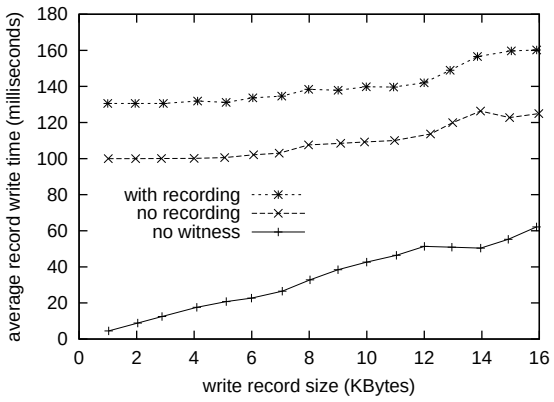


Figure 7: Record write time vs record size. The round-trip latency to the server and to the witness are both 100 ms. 160 KB are written using a selected record size and the average write time per record is computed.

function of the batch size (Section 6.1.2). With the naive or synchronous method and no recording, every SSL record has the witness overhead of 100 ms, which is equal to the witness latency in these experiments. With recording, there is an additional 31 ms disk access time per record. With the optimized or asynchronous method, the reply time does not increase linearly with the reply size. Instead, the shapes of the optimized witness lines are very similar to that of the no-witness line. Without recording, the pipeline bottleneck is negligible; therefore, the witness overhead without recording is nearly constant and is close to the witness latency. This is consistent with the estimate in Section 6.1.2. In fact, the performance overhead of a witness without recording is the same as that of a regular, non-caching HTTPS proxy that has a 200 ms round-trip latency to the client. The difference between the witness-with-recording overhead and the proxy overhead increases with the reply size, but at less than half of the estimated slope, *i.e.*, 9 ms per record vs. $2L_d$ =20 ms per record (Section 6.1.2). In practice, the recording can be done by asynchronous disk accesses; therefore, the witness overhead can be further reduced. The results confirm the analysis that the witness overhead for batched records is comparable to that of a regular, non-caching HTTPS proxy that is twice as far away from the client as the witness.

Figure 7 shows the SSL_write time for an SSL record as a function of the record size. In Section 6.1.2, we analyzed that the witness overhead per SSL record is independent of the record size, given that the disk write time is dominated by seek and rotation latency. The experimental results show that the overhead is actually decreasing as the record size gets larger. This is an artifact of the TCP buffering. The data passed to SSL_write() will not necessarily be transmitted over the network before SSL_write() returns. Therefore, an RPC to the witness might be overlapped with the actual transmission of the data in the previous SSL_write(). When the record size gets larger, the transmission time of a record gets more significant in comparison to the witness latency, which results in less perceived witness overhead. The maximum TCP buffer size in these experiments is 16 KB, while the total amount of data passed to SSL_write() in each experiment is 160 KB. Therefore, in the end of the write operations, at most 10% of the data was left in the buffer.

In summary, the experimental results confirm the analysis on the performance overhead of the witness, and show that the optimization with pipelining is effective in reducing the witness overhead.

# 7 Implications to other protocols

Intuitively, a security protocol between two parties A and B is likely a candidate for witnessing on behalf of the party A if it has the following properties:

1. The party B's identity is authenticated and the shared secrets are generated in a way that does not require the party A's secret or at least does not require non-decomposable computation with A's secret. This allows the witness to authenticate the party B's identity and generate the shared secrets on behalf of A, without knowing A's secret.

2. The encryption keys and MAC secrets used in the protocol are *independent* of each other. This allows the witness to give the encryption keys to the party A while keeping the MAC secrets to itself.

3. The protocol uses a *decomposable* MAC function, such as HMAC or NMAC [3], to provide message integrity. This allows the party A to compute and verify MAC's without knowing the MAC secrets.

An (infrequently used) option in SSL is for the server to request and verify the client's certificate. Since both the certificate and verification messages can be generated by the client without the involvement of the witness, and are independent of the rest of the handshake protocol, SSL is still witnessable on behalf of the client with this option turned on.

Another possible application of the witness is IPSec [10], a widely used security protocol at the IP layer. IPSec supports the Oakley Key Determination Protocol [14], which generates shared keys using the Diffie-Hellman key agreement method [16], and authenticates the identity of each party by signing some public information with that party's private key. It provides data confidentiality with encryption algorithms such as DES, and provides data integrity with the HMAC function. IPSec looks promising as a candidate for witnessing on behalf of either party. However, further details have yet to be worked out.

# 8 Related work

Related work includes non-repudiation services that aim to provide unforgeable evidence on the origins, contents, receipts, timed existence, *etc.*, of electronic messages.

The digital signature [18] of a message allows the receiver to prove the origin and contents of the message. Ideally, users of Internet transactions should be able to obtain receipts that are digitally signed by the servers. However, it requires the cooperation of the servers and implies the deployment of a *Public Key Infrastructure* (PKI). We know of few, if any, commercial web sites that digitally sign their web contents. Therefore, we believe that technologies that enable client-initiated evidence generation are valuable to today's Internet practices. Hopefully, such technologies will raise users' confidence, enable new or larger quantities of transactions to be done over the Internet, and eventually motivate the commercial web sites to adopt server-initiated approaches, such as digital signatures, to improve customers' experiences.

A certified mail delivery service guarantees that the receipt of an email message produces a receipt certificate whether the receiver is honest and diligent or not. Among the various classes of such services, the witness service is closest in spirit to those that use an online trusted third party during message exchanges [2, 23, 4, 1]. With the exception of [1], those services do not preserve message confidentiality from the trusted third party, and require the receiver and sometimes the sender to sign messages. The use of the trusted third party in [1] differs from the use of a witness in that both the receiver and the sender in [1] are aware of and need to cooperate with the third party, while in the witness scheme only the party that uses the witness service (*i.e.*, the client) cooperates with the witness.

A time-stamping service, or a digital notary service, provides unforgeable evidence on the existence of a digital document at a certain point in time, while preserving message confidentiality from the service provider [8]. However, it does not provide evidence for the origin of the document.

Overall, the witness service differs from existing non-repudiation services in the following two important ways: First, the witness service is mainly targeted at interactive transactions, though it can also be applied to static messages as a degenerate case; the existing non-repudiation services were designed for static or one-way messages, and their efficiency for interactive transactions has not been well studied. Second, the witness service is designed to work with existing security protocols (*e.g.*, SSL), while the existing non-repudiation services are based on new protocols between all involved parties and hence require all parties to either install new software or change their behavior.

# 9 Conclusion

We have presented a scheme to witnessing secure communication, using the novel algorithm of MAC decomposition. MAC decomposition allows the witness and the client to jointly compute the MAC without the witness knowing the message content or the client knowing the MAC secret. Therefore, the witness design meets the requirements of message confidentiality and evidence integrity simultaneously. Furthermore, the scheme requires no changes to the server, no changes to the communication protocol and only modular changes to the client, hence meeting the requirement of minimal changes.

We have implemented a prototype witness and demonstrated its robustness by using it to witness transactions with a variety of real web sites. We applied pipelining to minimize the performance overhead of witnessed communication and conducted experiments in a configurable emulated wide area network. The experimental results show that the performance overhead of a witness is comparable to that of widely deployed web proxies. Therefore, we conclude that the goals of evidence integrity, message confidentiality and manageable changes in the witness design are compatible with acceptable performance in practice.

We plan to open-source the witness code. Before we achieve that, the source code may be made available upon request.

## Acknowledgments

We would like to thank Martin Abadi, Mike Burrows, Stuart Haber, Ram Swaminathan and Bob Tarjan for giving valuable feedback, Dennis Fetterly for helping with experimental setup and Nitin Garg for porting the Mozilla browser source code.

# References

[1] M. Abadi, N. Glew, B. Horne, and B. Pinkas. Certified email with a light on-line trusted third party: Design and implementation. In *Proceedings of the 11th international WWW Conference*, pages 387–395, May 2002.

[2] A. Bahreman and J. D. Tygar. Certified electronic mail. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, pages 3–19, San Diego, CA, February 1994.

[3] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Advances in cryptography – Crypto '96 proceedings, Lecture Notes in Computer Science*, volume 1109, pages 1–15. Springer-Verlag, 1996.

[4] R. H. Deng, L. Gong, A. A. Lazar, and W. Wang. Practical protocols for certified electronic mail. *Journal of Network and Systems Management*, 3(4), 1996.

[5] T. Dierks and C. Allen. *The TLS Protocol, RFC 2246*. Internet Engineering Task Force, January 1999.

[6] FreeS/WAN: An implementation of IPSEC and IKE for Linux. http://www.freeswan.org.

[7] A. O. Freier, P. Carlton, and P. C. Kocher. *The SSL Protocol Version 3.0*. Netscape, November 1996.

[8] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991.

[9] International Telecommunications Union (ITU-T). *Recommendation X.509: The Directory-Authentication Framework*, 1989.

[10] S. Kent and R. Atkinson. *Security Architecture for the Internet Protocol, RFC 2401*. Internet Engineering Task Force, November 1998.

[11] R.T. Morris. A weakness in the 4.2BSD Unix TCP/IP software. Technical Report 117, Bell Labs, 1985.

[12] National Institute of Standards and Technology. *Secure Hash Standard*, April 1995. FIPS PUB 180-1.

[13] M. Oehler and R. Glenn. *HMAC-MD5 IP Authentication with Replay Prevention, RFC 2085*. Internet Engineering Task Force, February 1997.

[14] H. Orman. *The OAKLEY Key Determination Protocol, RFC 2412*. Internet Engineering Task Force, November 1998.

[15] Mikulas Patocka. Links: The WWW text browser. http://links.sourceforge.net/.

[16] E. Rescorla. *Diffie-Hellman Key Agreement Method, RFC 2631*. Internet Engineering Task Force, June 1999.

[17] R. Rivest. *The MD5 Message-Digest Algorithm, RFC 1321*. Internet Engineering Task Force, April 1992.

[18] R. L. Rivest, A. Shamir, and L. M. Aldeman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), 1978.

[19] L. Rizzo. Dummynet. http://info.iet.unipi.it/˜luigi/ip_dummynet, 2001.

[20] D. Sax. DNS spoofing (malicious cache poisoning), 2002. Published via Global Information Assurance Certification Practical Repository, http://www.giac.org/practical/gsec/Doug_Sax_GSEC.pdf.

[21] Squid web proxy. http://www.squid-cache.org.

[22] A. C. Yao. Protocols for secure computations. In *23rd Annual IEEE Symposium on Foundations of Computer Science*, pages 160–164, 1982.

[23] J. Zhou and D. Gollmann. Certified electronic mail. In *Proceedings of Computer Security–ESORICS '96*, pages 160–171. Springer Verlag, 1996.

# A Background

In this section, we give some background on the Secure Sockets Layer (SSL) protocol and on iterated one-way hash functions used to construct Message Authentication Codes (MACs).

## A.1 SSL

The SSL protocol allows client/server applications to communicate over the Internet in a way that prevents eavesdropping, tampering, and message forgery. SSL communication occurs through sessions and connections.

To establish an *SSL session*, the client invokes the *SSL Handshake Protocol* to negotiate with the server the cryptographic algorithms to use in the session, to optionally authenticate the server's identity, and to generate a shared secret between the two parties. The client provides a list of acceptable cipher suites and the server chooses one of them. The server typically identifies itself by sending an X.509 certificate [9] that contains the server's domain name and RSA public key [18], and is signed by a certification authority.

Assuming this typical case and that the client accepts the certificate—which sometimes happens automatically in some browsers—the client then generates a random number, encrypts it under the server's public key, and sends the result to the server. The server then uses its private key to decrypt the random number. The random number can then be converted to a shared secret between the two parties. The ability to compute the shared secret proves that the server is the party who owns the private key corresponding to the public key in the server's certificate. In this way, the server is authenticated to the client.

An SSL session consists of one or more *SSL connections*. Each SSL connection has connection-specific security parameters and is layered on top of a reliable transport protocol (*e.g.*, TCP). When a connection is created, the session's shared secret and the connection's additional parameters are used together to generate symmetric *encryption keys* and *MAC secrets* that are shared between the parties. Each direction of a connection has its own encryption key and MAC secret. The handshake that establishes an SSL session and the first connection in the session is called a *full* handshake, while the handshakes that create subsequent connections in an established session are called *partial* handshakes.

In an SSL connection, all control messages and application data are transmitted between the parties as *SSL records*. Each record is optionally compressed, has a MAC attached, and is encrypted before it is sent. The MAC consists of a key-dependent cryptographic hash of the record's content; a record with a correctly computed MAC can only have been generated by a party that has the appropriate MAC secret. In this way, attackers cannot spoof or undetectably alter SSL records. A received record is decrypted using the same key, its MAC is recomputed using the same secret and compared against the received MAC, and the record is decompressed if needed.

In order to prevent the dropping or replaying of messages by a third party, SSL gives each record a *sequence number*, which is also protected by the MAC. Each direction of a connection uses a separate sequence of numbers, starting from zero. When a party receives a record that does not have the sequence number it expected, a MAC error is reported and the session is invalidated.

In order to detect truncation attacks in which messages at the end of an SSL connection are dropped, SSL applications are required to shutdown each SSL connection with an explicit notification, called a close-notify message, to the peer.

The most popular application of SSL is the secure HTTP (HTTPS) protocol, which is used by most web sites that have privacy or security needs, such as financial sites and online merchants. It consists of plain HTTP running on top of SSL connections. Secure

FTP is another application of SSL.

## A.2 One-way hash functions and MACs

A *one-way hash function*, $H(m)$, operates on an arbitrary-length message, $m$, and returns a fixed-length hash value, $h$, with the following properties: (1) given $m$, it is easy to compute $h$; (2) given $h$, it is hard to compute $m$ such that $H(m) = h$; and, (3) given $m$, it is hard to find another message, $m'$, such that $H(m) = H(m')$.

An *iterated* one-way hash function, such as MD5 [17] and SHA1 [12], starts with a constant *initialization vector* (IV) and processes the message in blocks of fixed size—512 bits (64 bytes) for both MD5 and SHA1—and uses the result of the current block as the IV for the next block. That is, the hash of the first $i$ blocks of a message $(b_1, \ldots, b_i)$ is defined as

$$\begin{aligned} h_0 &= C \\ h_i &= H_{h_{i-1}}(b_i), \quad i > 0 \end{aligned}$$

where $H_v(b)$ denotes the result of applying an underlying hash function to both IV $v$ and message block $b$, and $C$ denotes the iterated hash function's starting IV. The hash of an entire message $m = b_1 + \cdots + b_n$ is then

$$H(m) = H(b_1 + \cdots + b_n) = h_n$$

where $b_1$ through $b_n$ are the sequence of blocks that $m$ is broken into and "+" denotes concatenation. The size of the IV and the result is 16 bytes for MD5 and 20 bytes for SHA1. If necessary, padding is added to the last block $b_n$ of the message to ensure that it is a complete block. We will use $H_v(m)$ to denote the result of applying the iterated hash function $H$ with starting IV $v$ instead of $C$ to message $m$. Thus, for example, provided $b$ is a complete block and $m'$ is non-empty, we have that

$$H(b + m') = H_{H(b)}(m')$$

A MAC function is a function that computes a one-way hash from a secret and a message. HMAC [3] is a particular way of constructing a MAC function from an iterated one-way hash function. HMAC and its variants are widely used in standard security protocols, including SSL 3.0, TLS 1.0 [5] and IPSec [10]. For example, the MAC of an SSL 3.0 record is defined as

$$\begin{aligned} &\text{HMAC}(sec, seq, l, content) = \\ &\quad H(sec + pad_2 + \\ &\qquad H(sec + pad_1 + seq + l + content)), \end{aligned}$$

where $sec$ is the MAC secret, $seq$ is the sequence number of the SSL record, $l$ is the length of the record, $content$ is the record content, $H$ is the underlying hash function, $pad_1$ is the byte $0x36$ repeated 48 times, and $pad_2$ is the byte $0x5c$ repeated 48 times [13].

HMAC uses the underlying hash function twice for better security. The use of $pad_1$ and $pad_2$ ensures that the first block fed to the hash function $H$ always consists of the MAC secret and padding only, which is independent of the message. This is a performance optimization technique, since the hashes of the first blocks can be computed only once and reused for multiple messages.

# B Sending and receiving an SSL record

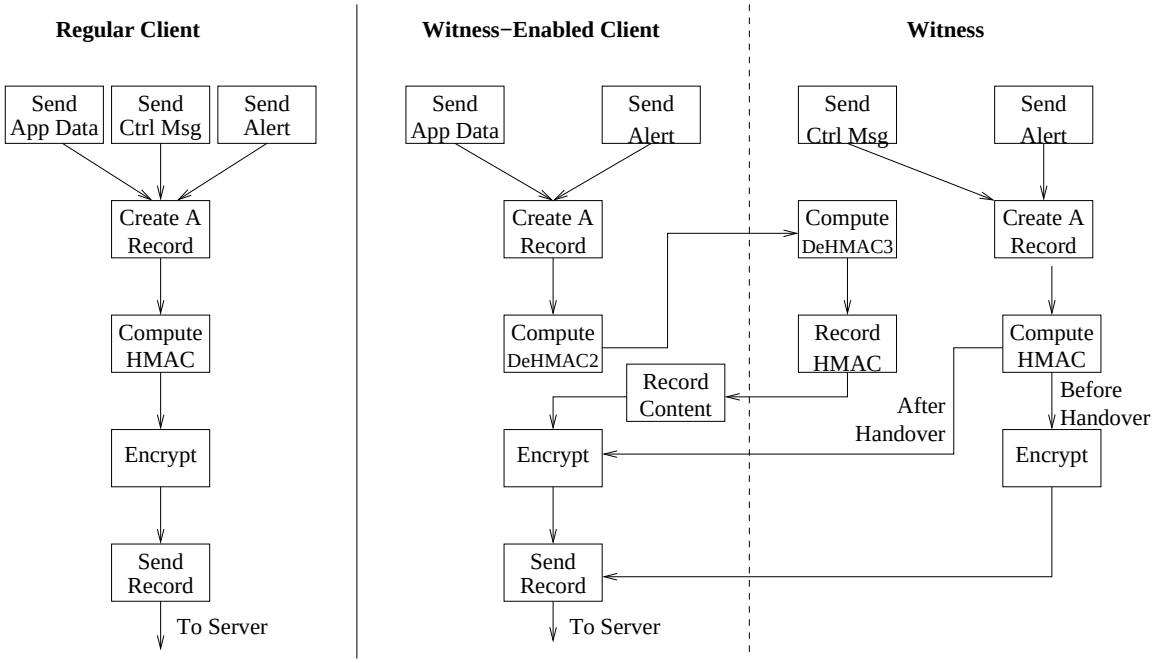


Figure 8: Sending an SSL record from the client or witness to the server. "Ctrl msg" in this figure refers to a handshake message or a change-cipher-spec message, not an alert message.

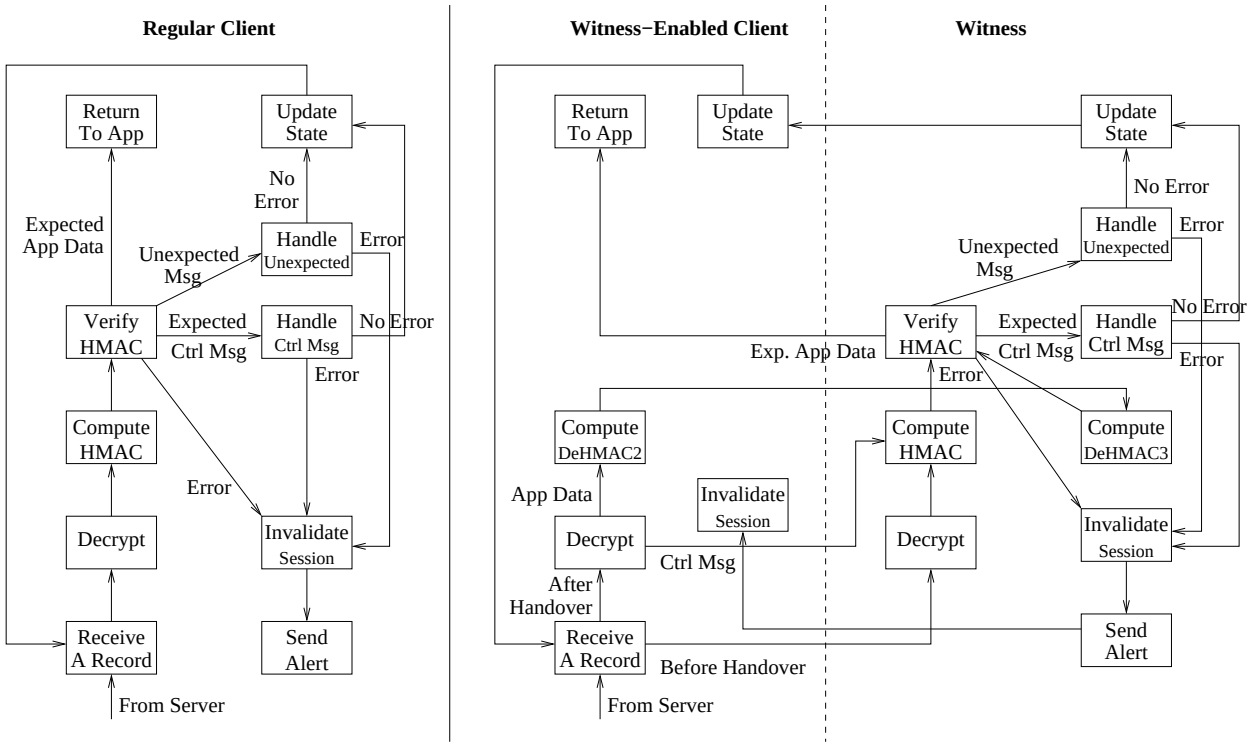


Figure 9: Receiving an SSL record from the server. "Ctrl msg" in this figure refers to a handshake message, a change-cipher-spec message, or an alert message. Recording of the MAC and the content, which is similar to that in Figure 8, is omitted in this figure for readability.