A Secure and Private System for Subscription-Based Remote Services

Pino Persiano Ivan Visconti

Dipartimento di Informatica ed Applicazioni
Università di Salerno
via S. Allende, 84081 Baronissi (SA), Italy

Abstract

In this paper we study privacy issues regarding the use of the SSL/TLS protocol and X.509 certificates. Our main attention is placed on subscription-based remote services (*e.g.*, subscription to newspapers and databases) where the service manager charges a flat fee for a period of time independent of the actual number of times the service is requested.

We start by pointing out that restricting the access to such services by using X.509 certificates and the SSL/TLS protocol, while preserving the interests of the service managers, neglects the right to privacy of the users.

We then propose the concept of a *crypto certificate* and the Secure and *Private* Socket Layer protocol (SPSL protocol, in short) and show how they can be used to preserve user privacy and, at the same time, protecting the interests of the service managers. The SPSL protocol only requires the user to have a *standard* X.509 certificate (with an RSA key) and does not require the user to get any special *ad-hoc* certificate.

Finally, we show the viability of the proposed solution by describing a system based on SPSL for secure and private access to subscription-based web services. Our implementation includes an SPSL-proxy for a TLS-enabled web client and a module for the Apache web server along with administrative tools for the server side. The system has been developed starting from the implementation of an API for the SPSL protocol that we describe in the paper.

Contents

List of Figures

1 Introduction

The main motivation of our work is the study of user privacy issues in subscription-based remote services. In a subscription-based remote service, a user is charged a flat fee for a period of time independent of the actual number of times the service is requested. The main concern of the service manager is to make sure that only customers that have paid the fee for the current period are granted access to the service. To do this, the service manager might give each user a username and a password to be used for accessing the service. In a more sophisticated approach, during the service enrollment phase, each user shows an X.509 [20] certificate to the service manager. The list of certificates shown by users at the enrollment constitutes the list of *qualified certificates*. The owners of the qualified certificates (*i.e.*, users that have access to the private key of a qualified certificate) are the *qualified users*. Each time a user requests the service, an SSL/TLS [12, 17] session is started. As part of the handshake protocol of SSL/TLS the user hands a certificate to the server and proves to be the legitimate owner of the certificate. Then, the server application matches the certificate against the list of qualified certificates and decides whether to grant access.

Our work deals with possible violations of user privacy within the context described above. We show that the practice of identifying users via certificates (or passwords), while preserving the rights of the service managers to screen users, neglects the right to privacy of the users. This is a paradigmatic example of two conflicting interests: the service manager needs to identify users to restrict access only to qualified users and users seek to protect their privacy.

As possible countermeasures we propose the concept of a *crypto certificate* and the *Secure and Private Socket Layer* protocol. Our aim is to provide a framework in which users can hide their identity and thus achieve anonymity while at the same time the service manager can be guaranteed that only qualified users gain access to the service. We also show that our proposal inter-operates with established standards (SSL, TLS and PKIX) by describing the implementation of a private navigation system based on a standard browser and web server.

1.1 Contributions of the paper

In this section we overview the contributions of our research.

1.1.1 Pointing-out the threats

We start by identifying three potential threats to the privacy of the users when X.509 certificates are used along with the SSL/TLS protocol.

The Secure Socket Layer (SSL in short, see [17]) proposed by Netscape Communications Corporation and its evolution, the Transport Layer Security (TLS in short, see [12]), used within the framework of PKIX [20] (a PKI based on X.509 certificates) promise to become the standard tool for implementing access control over the Internet. SSL and TLS allow a service manager to securely identify users requesting access to the service and to grant or to deny access according to an access control policy. In the simplest case, the access control policy consists of a list of qualified users that are to be granted access to the service. During the registration, each user gives his certificate to the service manager. For the moment, let us think of a certificate as consisting of the name of the owner, his public key and a validity

period signed by a trusted authority (called the *Certification Authority*). The certificate is exhibited by the user each time he requests access to the service. After verifying that the certificate is valid (*i.e.*, it has not expired or has been revoked and that the signature of the Certification Authority is valid) the manager can match the name on the certificate against a list of qualified users. If the certificate corresponds to a qualified user then the client is requested to give a proof that he knows the private key corresponding to the public key of the certificate presented (roughly speaking, this is achieved by exhibiting a digital signature of the hash of the transcript of the conversation up to this point). After that, a secure (*i.e.*, encrypted and authenticated) channel is established between client and server. This approach guarantees that an unqualified user cannot gain access to the service (unless, of course, he has access to the private key of a certificate corresponding to an authorized user).

As it is obvious by looking at the protocol specification, in SSL/TLS certificates travel in clear over the network. This follows from the consideration that the certificate by itself (*i.e.*, without the corresponding private key) cannot be used for impersonating the user. However, we point out that an eavesdropper could intercept all communication reaching a given service and build a list of users that access the service and compute the access frequencies of the users.

A second and more serious threat comes from the service manager himself. This is best explained by considering the following example. Users pay a monthly fee to access a medical publication database. The database contains information regarding various diseases. Each user is required to present his certificate in order to gain access to the database, and thus the manager can discriminate between users that have paid the monthly fee and users that have not. However, presenting the certificate allows the service manager to collect information on who has read articles about which disease. This information is not essential to the role of the service manager which consists in making sure that only qualified users access the database. The alternative solution of providing each user with a username and a password to be presented to the service manager using an SSL/TLS connection suffers the same drawback. Even though this approach dispenses with the need of client certificates, the service manager can still link requests to users which is exactly want we want to avoid.

The third type of attack to user privacy we envision is strictly linked to the structure of a certificate. Current standards for certificates profiles [20] give the possibility to add a list of attributes (called *extensions*) regarding the public key itself (*e.g.*, the Key Usage extension defines the purpose of the key) to the certificate. The certificate is made valid by the Certification Authority that signs the public key, the identity of the owner and the extensions. It is expected that new extensions will be introduced as the use of certificates spreads. Netscape Communications Corporation has already proposed some extensions to offer specific services to users of Netscape Navigator (*e.g.*, access to the policy regulating the issuing of the certificate at hand). Possible future applications come from the health care domain. In some situations people with a chronic disease benefit of exemption for some specific prescriptions related to their condition. So it is conceivable to have an extension in the certificate of the person carrying this information.

However, we believe that, although convenient, having too much information on the certificate is dangerous for the user's privacy. A person with diabetes uses his certificate to buy drugs on the web (in which case the extension specifying his condition is relevant) and to access the company web server (in which case the extension is not only irrelevant but giving

this information might delay or endanger a promotion). In general, each time the certificate is exhibited the whole set of information about the individual is given away, even though some pieces of information are irrelevant, and it is a very simple task to log this information into a dossier. This is only superficially similar to the case of personal ids that carry, besides first and last name, also the address, date and place of birth and current job. Showing the id reveals all the information that, in principle, could be logged but in practice it is very difficult to write down the information from a personal id that is shown to a security officer.

Notice that, with the current format of X.509 certificates, it is not possible to reveal just some of the extensions of a certificate (*e.g.*, only the extensions relevant to the transaction being performed) as the certificate is signed as a whole by the issuing authority. Therefore, the issuing authority signature can only be verified if the whole content of the certificate is disclosed.

1.1.2 Shared password

Before describing our proposal, let us point out the weaknesses of a very simple solution: giving all legitimate users the same *shared* password. In this way, the privacy of the user is protected (the server receives the same password independently of the user that has requested the service) and the service manager is guaranteed that only legitimate users access the service. Moreover, it seems to dispense with the need of public keys on the client side and thus of the necessary infrastructure. A first weakness of this approach is that a single shared password is very difficult to revoke. The only way to revoke a shared password is to pick a new password and securely broadcast it to all the legitimate users. Besides requiring huge communication and forcing the users to keep track of the passwords, it is not clear how the new password can be securely communicated to the users. The most obvious way would be to send it by postal mail or to send it encrypted by e-mail. The first solution does not seem practical whereas sending password by encrypted e-mail requires the users to have public keys. On a different note, we stress that users have no guarantee that the shared password is the same for all the users. Indeed, a malicious server might pick a different password for each user and thus link service requests to users. As a final remark, we would like to point out that all password-based schemes are very conveniently abused by users that *lend* their password to other users. On the contrary, schemes like ours that employ digital certificates make the sharing of subscriptions much more inconvenient. Indeed, in order to share his own subscription, one has to trust his private key to other users thus restricting the practice of sharing to the same household (which is considered acceptable by most service managers). On the other hand, if a user wishes to allow "friends" to access the service using his own subscription and does not want to give away his private key, then he could set up a proxy service that gains access to the service on behalf of the "friends." Obviously, this requires technical knowledge beyond the one of the typical user and is much more inconvenient than just lending a password.

1.1.3 Accountability and anonymity

The controversy about user accountability rises often within the context of anonymity and is somehow beyond the technical aspects of our research. Indeed one wonders that if users are shielded by perfect anonymity what is to prevent them from abusing the system? We believe

that this is the wrong question to ask if one considers the privacy of the user to be a primary goal as accountability (and escrow) requires to place trust in a third party. Rather we think that one should design protocols so that they cannot be abused by fraudulent users. In the specific case of subscription-based services, accountability is often justified as a way to detect abnormal user activity which points to subscription sharing. We prefer to address the problem in a different way (see the discussion above) and thus dispense with the need of a trusted third party.

1.1.4 Preserving the privacy

In this paper we present the *Secure and Private Socket Layer* protocol (SPSL protocol, in short). The SPSL protocol is an extension of the SSL/TLS protocol that allows the server to present a list of certificates corresponding to qualified users and the user to prove that he *"knows"* the private key associated with at least one of those certificates without revealing which one. The technical core of our proposal is constituted by a simple protocol for proving knowledge of one private RSA key out of a set of l keys and is derived from results of [11]. Our approach protects the right of the service manager to give access to his service only to those who have paid for it and at the same time protects the right of the user to preserve his privacy. The service manager can still collect statistics about which piece of information has been accessed by each request but cannot link requests to users.

We also propose a modification of the structure of the certificates so that partial disclosure of information is possible. Our proposed modification still allows to verify the integrity of the certificate but gives the user the possibility of disclosing only some of the extensions. At the same time, it is not possible for the user to modify the value of the extensions shown.

For example, going back to the example above, when buying drugs the user discloses the extensions related to health-care whereas when accessing the company web he discloses the extensions related to his job. In both cases, the service manager is guaranteed that the extensions disclosed have not been forged and obtains no information about the undisclosed extensions.

1.1.5 Design and implementation of a private navigation system

To test the viability of our approach, we have built a private navigation system based on SPSL. We started by developing an SPSL API so that upgrading existing SSL/TLS applications to SPSL can be achieved with very little effort. The resulting applications can transparently inter-operate with SSL/TLS and SPSL applications as in our implementation we have used cipher suites codes reserved for experimental algorithms (those with the first byte equal to *0xff*, see Appendix A.6 of [17]). Building on this, we construct SPSL-enabled HTTP browser and server. More precisely, on the server side, we have developed an SPSL module for the HTTP server Apache [1]. The module is based on the widely used ModSSL module [21]. The SPSL Apache module provides directives for the web site administrator to specify access restrictions to the resources. The web site administrator builds the access lists associated with the restricted resources of the site; access lists specified by the list of qualified certificates are published with mime type `application/x-x509-mask`. Apache servers supporting the SPSL protocol are fully compatible with SSL/TLS compliant browsers. Details are presented in

Section 3.6 and in Section 5.3.1.

On the client side, we have designed and implemented an SPSL proxy application for Netscape Navigator thus making SPSL usable from a standard browser. The communication between the proxy and the browser uses SSL/TLS. To make the proxy transparent to the user, we have the proxy use the same certificates used by the browser for SSL/TLS. This implies, among other things, that the user needs not to request an *ad-hoc* or an "SPSL-aware" certificate.

The SPSL proxy supports the mime type `application/x-x509-mask` to exchange the set of certificates corresponding to the users qualified to access a resource. This allows to cache the certificates thus dispensing with the need of exchanging the set of qualified certificates each time the resource is requested. Details are presented in Section 5.3.2.

In Section 5, we give the skeleton of the client and server of an application that employs SPSL. In Section 5.3.1 we present the new directives for the SPSL-compliant web server Apache. In Section 5.3.2 we present the features of the SPSL proxy application.

Even though we have used our API only to developed a private navigation system, our API has been designed to be a general substitute for the TLS API and can thus be used as a basis to develop other private applications.

All the software developed and the documentation regarding configuration options are publicly available at [30].

1.2 Related work

A fair amount of research has traditionally centered around anonymity starting with early works by Chaum [9]. The need for anonymity has been made more pressing with the widespread of the Internet. It is well-known that the Internet provides very minimal security and privacy and it is a trivial task to monitor and record many of the electronic actions taken by a user. This information can be used for targeted advertising or more intrusive purposes.

Some of the solutions to the problem of anonymity provided so far have concentrated on the problem of hiding the IP address (or, more generally, network-level information) of the user; *e.g.*, Mix networks [9, 19], Onion Routing [25], and CROWDS [26]. However, these schemes cannot be easily adapted to work within the context of subscription-based remote services. For example, CROWDS [26] is a system that makes it very difficult for a web server to traceback the requester of a resource. The basic idea is that a user should blend into a crowd before accessing the web. The web server upon receiving the request knows only that the request originated from a member of the crowd but then it is difficult for the web server to check whether the user is qualified to access the service. Rather, we consider systems that hide network-level information like CROWDS [26], Mix networks [9, 19] or Onion Routing [25] as complimentary to ours.

Group signatures (introduced in [8]; see [3] for an overview) is another concept related to anonymity. A group signature scheme allows a member of a group to sign a message on the group's behalf in such a way that only a designated *group leader* can link a signature to the member of the group that has produced it. Among the several constructions presented in the literature the one by Camenisch [5] is the closest to our needs as it provides for very efficient procedures for adding and removing new users. The construction in [5] is based on the hardness of discrete logarithm and the core technical tool is a scheme for proving knowledge of

one discrete logarithm out of n (which makes it conceptually very similar to our construction derived from [11]). In contrast, our construction can be based on RSA making it compatible with real-world scenario in which X509 certificates are used. Moreover using a group signature scheme requires to place trust into the group leader that is useful for escrowing but is exactly what we wish to avoid. Even though the escrowing feature can be removed from some of the group signature schemes (for example, the one presented in [2]) this is not true in general. Finally, we stress that the issue of efficient revocation of users for group signatures has not been successfully addressed yet.

A very similar concept is the concept of a *ring signature* [28]. A ring signature scheme can be thought of as a group signature scheme with no group leader. The implementation proposed in [28] requires the signer to compute $l-1$ public key encryptions (which can be performed very efficiently with low-exponent RSA or Rabin's cryptosystem), one public key decryption and l symmetric encryptions (which again can be performed very efficiently), where l is the size of the group of possible signers.

Schechter *et al.* in [29] have proposed a scheme for anonymous authentication in dynamic groups that can easily inter-operate with X509 certificates. Here the server sends the same message m encrypted using each of a set of l keys. The client proves possession of the private key corresponding to one of the l keys by returning the original message m to the server. We observe though that, even if not specified by the protocol, once the message m has been obtained using the private key in his possession and before sending m back to the server, the client has to verify that the same message m has been encrypted by using the l keys. If this verification step is skipped, the server might pick a different message for each of the l keys and infer the identity of the client from the message received. Thus the protocol of [29], just like ours, requires the client to perform one decryption and $l-1$ encryptions. The main drawback of protocol of [29] is the additional cryptographic assumption made on the encryption scheme used: encrypting the same message with different public keys does not reveal the message. This is not true, for example, for the Rabin cryptosystem and for RSA with low exponents which, because of very efficient encryption algorithms, are ideal candidate cryptosystems. On the other hand, as it will be clear in the rest of the paper, our protocol does not need any additional cryptographic assumption.

In pseudonymous certificates [7] the fields used to identify the owner of a certificate are filled with random values in order to make the certificate unlinkable to its owner. However, we observe that in the context of subscription-based remote services the use of pseudonyms does not help as accesses by the same user can be linked and loss of privacy for external reasons in one access totally compromises the privacy of the user.

The concept of selective disclosure of private information has been addressed in [4] in which a different model of digital certificate is proposed. The solution presented is very elegant and powerful as a certificate is completely blind and the owner of the certificate can prove (without giving any additional information of the certificate) that the attributes of the certificate satisfy a certain formula (for example, "the owner lives in Europe and is older than 18"). The main drawback of this approach is that it is difficult to integrate with the public key infrastructure model of PKIX and TLS. In Section 4, we propose a new format for digital certificates that allows one to selectively disclose attributes of a certificate (but it is not possible to prove general statements as in [4]) and can be easily integrated with PKIX and TLS.

A different solution to the problem of anonymity in transactions has been presented in [31] and is based on the concept of an *unlinkable serial transaction* which can be implemented using *blind signatures*. Blind signature schemes, first introduced by Chaum [6], allow one party (the requester) to have a message signed by another party (the signer) with the signer receiving no information about the message that is being signed. Roughly speaking, in the scheme presented in [31], during the enrollment phase the user has the server sign an access token using a blind signature scheme. Each time the user wishes to access the service, it gives the server the signed token he has received from the previous transaction and receives a new blindly signed token from the server. The privacy requirement is seen to hold by the properties of the blind signatures. It is also easy to see that, for each user there is, at any given time, only one legitimate token thus forcing seriality of transactions. This has the advantage of making sharing of subscription very inconvenient as it requires the users to exchange the current valid token. On the other hand, the server has to keep a list of invalid tokens to avoid that the same token is spent more than once. From a more practical point of view, if a user wishes to access the service from multiple machines (*e.g.,* workstation, laptop, PDA) then he has either to transport the current valid token from machine to machine (which is seen as inconvenient by the user), or perform the enrollment phase once for each machine he wishes to use. Alternatively, a secure network storage for the token from which the current valid token can be retrieved must be available.

Our proposed protocol is based on the concept of anonymous group identification. A full and general solution to the problem of anonymous group identification has been given in [11], as an application of some structural results on the class of languages having perfect zero-knowledge proofs. Specifically, the results in [11] give perfect zero-knowledge proofs for any set of witnesses associated with a satisfying assignment of any monotone formula over random self-reducible languages (*i.e.*, RSA, quadratic residuosity modulo composite integers, graph isomorphism, membership to a subgroup modulo a prime, decision Diffie-Hellman problem). A protocol that is more efficient from the communication point of view is found in [10]. Unfortunately, this protocol is based on the hardness of the Quadratic Residuosity problem and cannot be adapted to work in a practical setting with X509 certificates but, instead, it requires *ad-hoc* certificates.

2 The basic anonymous identification protocol

In this section we present the anonymous identification protocol which is at the base of SPSL. For sake of concreteness, we consider the public keys to be RSA keys. We point out that, under very general assumptions, the same protocol can be used if some other public-key cryptosystem is considered.

Consider the following two-party game: manager M has a list of RSA public keys: $(N_1, e_1), \cdots, (N_l, e_l)$ and user U wishes to convince M that he knows the secret key corresponding to one of the public keys without revealing which one he knows. The manager sends a random message m to the user asking for the signature of l messages (one for each public key) such that the XOR of the l messages is equal to the original message m. In SPSL, the message m is not picked by the manager but is instead a hashed version of the transcript of the messages exchanged so far. A detailed description of the protocol is found in Figure 1 and implementation details are discussed in Section 5.1.

Common Input: l RSA public keys: $(N_1, e_1), \cdots, (N_l, e_l)$ of length n.

U**'s Private Input:** (i, d_i) such that $d_i \cdot e_i \equiv 1 \bmod \phi(N_i)$.

Instructions for Manager and User:

M.1 Pick a random message $m \in \{0,1\}^n$.
Send m to U.

U.1 For each $1 \leq j \leq l$ and $j \neq i$

U.1.1 Pick a random *signature* $s_j \in Z^*_{N_j}$.

U.1.2 Compute $m_j \equiv s_j^{e_j} \pmod{N_j}$.

U.2 Compute $m_i = m \oplus m_1 \oplus \cdots m_{i-1} \oplus m_{i+1} \oplus \cdots m_l$.
Compute $s_i \equiv m_i^{d_i} \pmod{N_i}$.
Send $(m_1, \cdots, m_l, s_1, \cdots, s_l)$ to M.

M.2 Verify that $m_1 \oplus m_2 \oplus \cdots \oplus m_{l-1} \oplus m_l = m$.
For $j = 1, \ldots, l$ if $s_j^{e_j} \not\equiv m_j \pmod{N_j}$ then ABORT.

Figure 1: The basic anonymous identification protocol

Next we discuss the properties of the protocol. By inspection it is easy to see that if the user has the private key corresponding to one of the l public keys then verification step M.2 is always successful. On the other hand, we observe that a user, which is not in possess of any of the private keys, has negligible probability of succeeding in computing the signature of l messages whose XOR gives the message m. For example, if the user tries to sign random messages by choosing signatures s_j at random then the probability that the XOR of the l messages is the encryption of a specific message is about 2^{-n} (n is the key length). We also remark that using standard techniques it is possible to modify the above protocol (at the expenses of increased computational and communication costs) so that any user U that succeeds in passing the manager's verification step possesses at least one private key.

Let us argue that the manager does not learn which key is known to the user. Consider two users U_1 and U_2 and suppose U_1 knows key i_1 and U_2 knows key i_2 and for each message m let us look at the distributions $D_1(m)$ and $D_2(m)$ induced by U_1 and U_2 on the message sent to the manager. It is easy to prove that distributions $D_1(m)$ and $D_2(m)$ coincide and thus the protocol is perfectly witness indistinguishable [16]. Using a standard technique and one extra round of communication the protocol can be turned into a perfect zero-knowledge protocol [18]. Notice that the protection of the user's privacy is unconditional and thus privacy of an access is not lost even if the user's private key is later compromised. Thus our scheme enjoys the *perfect forward anonymity* property.

As already pointed out in the introduction our protocol is *fully dynamic*: adding and removing users from the set of qualified users does not affect other users. Indeed, the set of qualified users is determined by the set of l public keys. If a users leaves (*e.g.*, he has not paid the fee for the current period) or a new user enrolls all the manager has to do is to update the list of public keys. The keys of the old users do not change and can still be used to get access to the service. The only difference a user will see is in the different set of public keys that will be involved in the protocol.

3 The SPSL protocol

In this section we describe the SPSL protocol that is meant to be the privacy enhanced version of SSL/TLS. The SPSL protocol allows the client to securely access a service and to identify himself as a qualified user (so far it is similar to SSL/TLS) without revealing his identity (this is the added feature of SPSL) using the basic anonymous identification protocol described in the previous section. To specify the protocol, we use the same syntax as in the specification of the TLS protocol [12]. We also point out that TLS is a special case of SPSL (corresponding to the case in which anonymity is not desired) and in the description we only focus on the newly added features.

3.1 Background on TLS

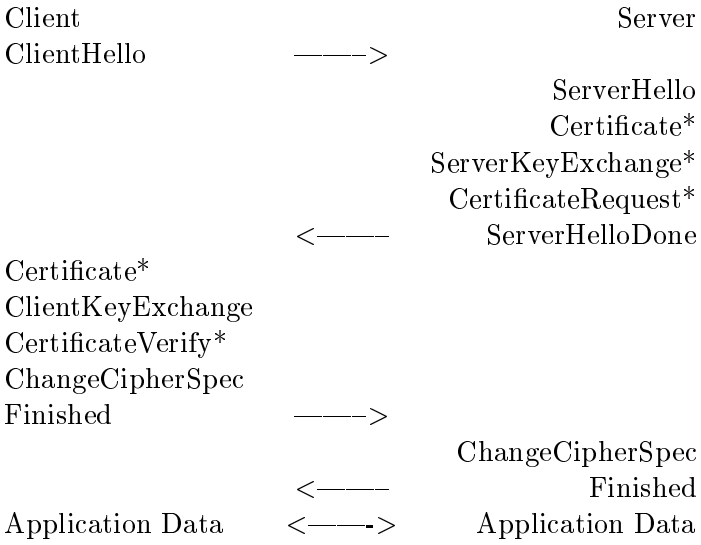
The TLS protocol (derived from the SSL protocol proposed by Netscape Communications Corporation, see [17, 12]) is the de-facto standard for secure communication over the Internet. It guarantees secure message exchange, authentication and secure identification. The SSL/TLS protocol consists of different protocols. Our work is concerned with the Handshake Protocol which we now briefly review.

During the Handshake the two parties negotiate the *cipher suite* to be used for the connection. The cipher suite specifies the algorithms to be used to exchange the master key, to perform identification, to encrypt application data and to compute the MAC of the data. The negotiation is performed by the client that sends a *ClientHello* message containing a list of cipher suites; the server must respond with a *ServerHello* message containing the cipher suite he chooses among the one proposed by the client. The SSL/TLS API developed as part of the OpenSSL Project [23] includes the `SSL_set_cipher_list` function that can be used to specify the preferred cipher lists. Typically the server picks (and sends as part of the *ServerHello* message) the first cipher suite of his lists that appears in the *ClientHello* message. The *ClientHello* and *ServerHello* also establish the following attributes: Protocol Version, Session ID, and Compression Method. Additionally, two random values are generated and exchanged: ClientHello.random and ServerHello.random that will be used to generate the session keys. Following the hello messages, the server at this point sends his certificate using the *Certificate* message[1]. If the certificate sent by the server does not contain an encrypting key a *ServerKeyExchange* message containing parameters for a key exchange protocol (*e.g.*, Diffie-Hellman parameters) is sent.

If the access control policy of the server requires client authentication, then the server sends a *CertificateRequest* message to which the client then replies with a *Certificate* message that contains an X.509 certificate. The server checks the validity of the certificate (*i.e.*, verifies that the signature of the issuing authority is correct) and then asks the client to "prove knowledge" of the private key associated with the public key of the certificate. This step is necessary as certificates are public and just holding a certificate does not prove your identity. At this point, the client and server could engage in a zero-knowledge proof of knowledge [18, 15], but for efficiency reasons, the "proof of knowledge" is replaced by a signature of a hashed version of the transcript of the conversation so far which is sent as part of the *CertificateVerify* message.

[1] This message is actually optional as the server might choose not to authenticate himself by giving the certificate. However, in this case the connection is subject to the man-in-the-middle attack.

The rationale behind this choice is that it is very unlikely that a client not possessing the private key could succeed in exhibiting such a signature and that, since the message being signed is somewhat random, the server does not get any information about the private key of the client. At this point, a *ChangeCipherSpec* message is sent by the client followed by the *Finished* message under the newly established cipher suite. In response, the server sends his own *ChangeCipherSpec* message followed by the *Finished* message under the new cipher suite. Moreover the client sends a *ClientKeyExchange* containing data for the key exchange algorithm (if the server has exhibited a certificate with an encrypting key, then this message consists of a random seed encrypted with the server's key from which various sessions keys are derived). At this point, the handshake is completed and the client and server may begin to exchange application layer data (see Figure 2).



(*) Optional message.

Figure 2: The SSL/TLS Handshake Protocol

3.2 The SPSL protocol overview

SPSL differs from SSL/TLS in the handshake phase in which the client proves knowledge of the private key associated with the certificate he has exhibited. Once the handshake has been performed, SPSL establishes a secure and authenticated channel between the client and server (just like SSL/TLS).

During the SPSL handshake, one of the two parties (in the simplest case the server, but see Section 3.6) exhibits a list of l certificates (we call such a list the *mask* as it masks the actual certificate held by the client), and then the client proves knowledge of at least one private key that corresponds to one of the l public keys. This is done using the basic protocol presented in Section 2 (see Section 5.1 for implementation details). The mask consists of the certificates belonging to the set of qualified users. One of the issues we have to deal with is related to

the fact that different services or resources managed by the same server might be associated with different masks. As an example, consider an HTTP server. The various resources (be it a simple HTML file or a service in the form of a CGI application) managed by the server could be associated with different sets of qualified users. As SPSL lives at a lower level (session level) than HTTP (which is at the application level), the actual command to get the resource is issued as part of the application level data and only after the handshake protocol has been completed. Thus it could be the case that the handshake has been performed with a mask that is inappropriate for the resource requested by the client and has to be performed again. To avoid the double handshake, the server publicizes (see Section 3.6) the masks associated with the various resources managed. In this case, first the client identifies the appropriate mask for the resource he is interested in (again, see Section 3.6 on how this is achieved) and then, as part of the SPSL handshake, he submits the mask to the server. Once the handshake is completed and the request is actually performed by the client (*e.g.*, by issuing the HTTP GET command), the server verifies whether the mask used is appropriate (*i.e.*, it consists of a subset of the set of certificates associated with the resource) for otherwise the anonymous identification protocol has to be performed again with the right mask (see the discussion regarding function `SSL_access_control_verify` in Section 5.2).

3.3 The SPSL cipher suite

Similar to TLS each SPSL connection is associated with a cipher suite that specifies the actual cryptographic algorithms to be used. An SPSL cipher suite consists of the following components:

1. a key exchange algorithm, an identification algorithm, a bulk encryption algorithm (including secret key length) and a MAC algorithm;
2. the anonymity level (this is the only component of an SPSL cipher suite that does not appear in a TLS cipher suite).

For example, the SPSL cipher suite RSA-RC4-SHA-STRONG uses RSA for key exchange and identification, RC4 to encrypt application data, the SHA algorithm as message digest, and specifies the STRONG (see below) level of anonymity.

The following anonymity levels have been introduced:

NULL This is the anonymity level of SSL/TLS; the cipher suites with anonymity level NULL can be used for compatibility with the SSL/TLS protocol.

WEAK When this anonymity level has been selected, the server may ask for a client certificate in which case the client sends his certificate encrypted with the server's public key. The server's certificate must have an encrypting key. WEAK anonymity addresses the first privacy concern discussed in Section 1.1 that considers an eavesdropper intercepting the incoming traffic of the server.

MEDIUM A set of certificates each with an encrypting key (or an identifier of a set of certificates, see later for more details) is proposed by either the client or server. The client then proves knowledge of the private key associated to one of a set of certificates using the protocol of Section 2. MEDIUM anonymity addresses the second privacy concern discussed in Section 1.1 and protects the user from the service manager.

STRONG This is similar to the MEDIUM level with the exception that if the set of certificates (or its id) is proposed by the client then it is sent encrypted using the server's public key.

3.4 The structure of SPSL messages

In this section we specify the structure of the messages introduced by SPSL and how the SSL/TLS message types are to be modified. We do not repeat here the definition of the TLS data types that can be found in [12].

We start with the *CertificateRequest* message that is used by the server to request the certificates. A definition of the *CertificateRequest* message using ASN.1 [13] is found in Figure 3. There, *ID* is an identifier for a list of certificates, *mask_list* is a list of Certificate and ASN.1Cert is a sequence of bytes (see Section 5.6.2 of [17, 12]).

```
struct {
  select (PrivacyLevel)
    case NULL:
      ClientCertificateType certificate_type⟨1...2^8-1⟩
      DistinguishedName certificate_authorities⟨3...2^16 − 1⟩
    case WEAK:
      ClientCertificateType cert_type⟨1...2^8 − 1⟩
      DistinguishedName cert_authorities⟨3...2^16 − 1⟩
    case MEDIUM:
      opaque ID[4];
      ASN.1Cert ca_list ⟨0...2^24 − 1⟩
      ASN.1Cert mask_list ⟨0...2^24 − 1⟩
    case STRONG:
      opaque ID[4];
      ASN.1Cert ca_list ⟨0...2^24 − 1⟩
      ASN.1Cert mask_list ⟨0...2^24 − 1⟩
} CertificateRequest;
```

Figure 3: The definition of the *CertificateRequest* message.

The CertificateList message is used for connection with MEDIUM level of anonymity and its definition is found in Figure 4. Here, *subset* is an ordered sequence of 2-byte integers, each specifying the position of a certificate in the list *ID*. When *subset* is not empty, the proof of identity is based only on the certificates specified by *subset*. This feature of SPSL improves the performance of the protocol (see discussion in Section 5.3.3).

The *EncryptedCertificateList* is used when STRONG anonymity is sought and it has the following structure:

public-key-encrypted CertificateList *EncryptedCertificateList*.

The *CertificateVerify* message is described in Figure 5. The field *proof_of_identity* is the output of the procedure described in Section 5.1 on the same string that is signed in TLS (see Section 7.4.8 of [12]).

```
struct {
   opaque ID[4];
   uint16 subset⟨0...2^16 − 1⟩
   ASN.1Cert ca_list⟨0...2^24 − 1⟩
   ASN.1Cert mask_list⟨0...2^24 − 1⟩
} CertificateList
```

Figure 4: The definition of the *CertificateList* message.

```
struct {
   select (PrivacyLevel)
      case NULL:
         Signature signature;
      case WEAK:
         Signature signature;
      case MEDIUM:
         opaque proof_of_identity⟨0...2^24 − 1⟩;
      case STRONG:
         opaque proof_of_identity⟨0...2^24 − 1⟩;
   } CertificateVerify;
```

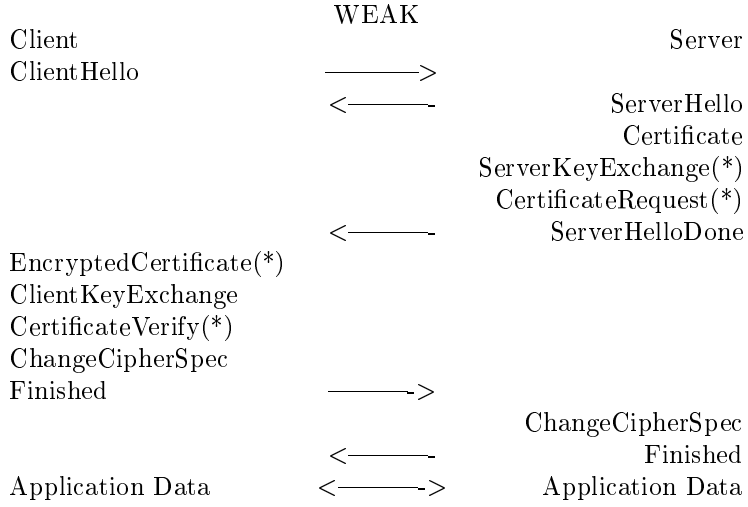
Figure 5: The definition of the *CertificateVerify* message.

3.5 The SPSL handshake protocol

The ClientHello and the ServerHello messages specify the anonymity level that parties have negotiated. The anonymity level determines the rest of the handshake. The next figures depict the exchange of messages for anonymity level WEAK, MEDIUM and STRONG. For the NULL anonymity level, the handshake messages are the same as those exchanged in the SSL/TLS protocol (see Figure 2).

The exchange of messages during the handshake for a connection with WEAK anonymity level is described in Figure 6. We stress that the client sends the *EncryptedCertificate* message and the *CertificateVerify* message only if the server sends the *CertificateRequest* message.

The exchange of messages in the case of MEDIUM level of anonymity is described in Figure 7. The *CertificateRequest* message carries a list of certificates with an encrypting public key or an identifier for such a list that describes the mask to be used for the connection. Either one or both can be empty. In the latter case, the server lets the client specify the mask to be used for the connection (see more on this in Section 5.2 in the discussion of the function `SSL_set_mask_list`). This is done by the client by sending a *CertificateList* message containing a list of certificates with an encrypting public key or an identifier for such a list. The client has access to the private key corresponding to one certificate. The certificates are presented in a random order so that no information is given about which private key is known by the



(*) Optional message.

Figure 6: The SPSL handshake for the WEAK level of anonymity.

client. The message can specify a subset of the list, in which case the client will perform his proof of identity only with respect to the specified certificates (see discussion in Section 5.3.3).

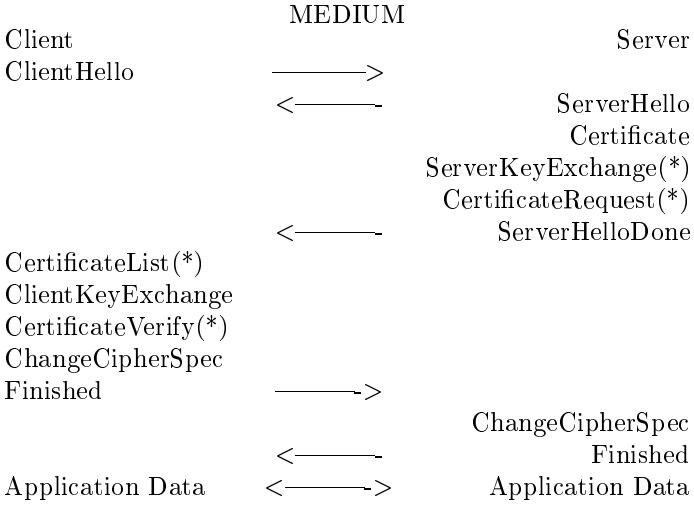
The exchange of messages for a STRONG connection is described in Figure 8. As it can be easily seen the messages exchanged are very similar with those exchanged for the MEDIUM level of anonymity. The only difference is that an *EncryptedCertificateList* message containing an encrypted certificate list is exchanged instead of a *CertificateList* message.

3.6 Specifying a mask by the identifier

As we have seen, in SPSL the list of qualified certificates for the required resource may be specified by the server by sending the complete list to the client. This has two drawbacks. First of all, the exchange of the certificates might constitute a communication bottleneck. Second, the server could send fake public keys along with a reasonably small number of real public keys. If the client passes the anonymous identification protocol the server obtains information on the identity of the client.

If a trusted party is available, we could follow the approach of [29] that requires a trusted third party to sign the list of qualified users each time a user is added or removed.

On the other hand, if no trusted party is available, we suggest that each server publishes the *mask list* of the service; that is, a sequence of sets of certificates each with its own identifier and the list of resources each with the identifier of the relative qualified set (the approach is similar to and inspired by the *Resource Description Framework* [32]). Once the mask list of the service has been published (and the client has downloaded it) there is no need to exchange it each time access is requested by the client. Furthermore, the service manager is committed to a list of qualified certificates and cannot change it in order to detect the identity of the client. To allow the mask list to change (by adding and removing users), each mask list could be identified by a serial number and the client could check, before the protocol starts, if a



(*) Optional message.

Figure 7: The SPSL handshake for the MEDIUM level of anonymity.

new mask is available from the service provider by simply requiring the serial number of the current list. If a new mask list is available then the client needs only to download the difference between the two mask lists.

For web applications, we have introduced the mime type

`application/x-x509-mask`

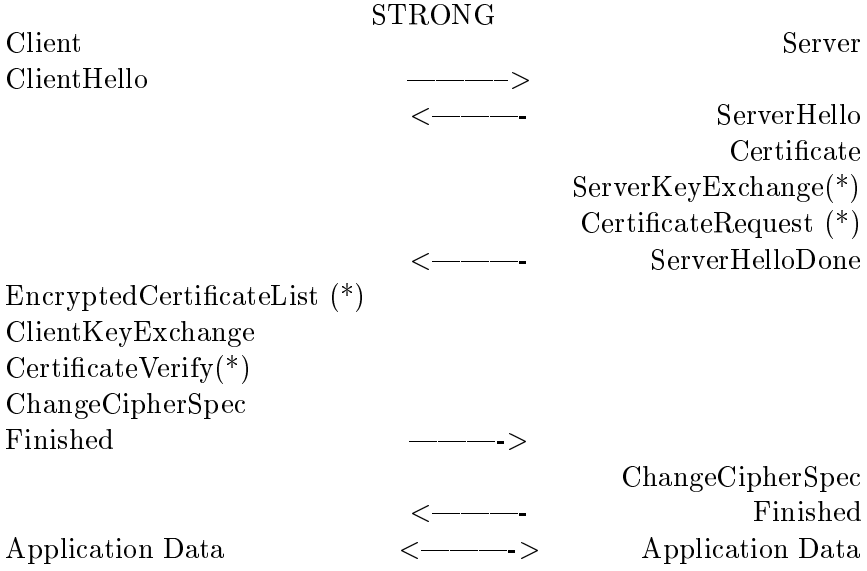
for messages carrying lists of masks.

An HTTP message with content-type `application/x-x509-mask` has the format specified in Figure 9. The first line carries the serial number of the mask list and the subsequent line (the one starting with `ID`) contains the name of the mask and its validity. Each line starting with `RES` reports the URI of a protected resource. Resource lines are then followed by the sequence of certificates in PEM format specifying the list of qualified users.

4 Crypto certificates for the selective disclosure of certificate extensions

In this section we discuss the concept of a *crypto certificate*. The owner of a crypto certificate can disclose only some of the extensions of a certificate and still the certificate can be verified by a second party. Notice that, although we will only discuss the use of crypto certificates within the context of SPSL, crypto certificates can be also used in conjunction with SSL/TLS to preserve the privacy of the users.

The certificate profile for X.509 certificates [20] specifies that the issuing certification authority signs the entire certificate using its own private key. The values of the extensions are provided by the user himself and there might be different policies as to how the truthfulness of the information provided by the user is ensured. Generally users have few digital certificates (often only one) because it is not practical to manage several certificates and Certificate Au-



(*) Optional message.

Figure 8: The SPSL handshake for the STRONG level of anonymity.

thorities usually charge for certificates and thus it is expected that one digital certificate could contain several extensions. Each time the user requests a restricted service and uses his digital certificate the whole set of extensions (even those not connected with the specific application in execution) are transferred to the service manager. The information gathered can then be used for example to construct a user profile.

Our solution to these problems is based on a new concept of digital certificate that we define *crypto certificate*. A crypto certificate is similar to a X509 certificate but some values of the extensions are not in clear format. To be more precise let us describe the life cycle of a crypto certificate. The user requests a certificate by issuing a *crypto request*. The user generates the crypto request by hiding each field that could damage his privacy by means of a secure hash function (see [14, 27]). For each such field, the user picks a random value of a fixed length and hashes the concatenation of the value of the field and the random padding. The value of the extensions in clear and random paddings used are submitted together with the hash value to the Certification Authority.

Before signing a *crypto request*, the certification authority verifies that the hash of the hidden fields and the associated random paddings have been correctly computed. The *crypto certificate* that is signed by the certification authority can be used by the user to disclose private information only when they are relevant for an SSL/TLS transaction in the following way. The user submits the crypto certificate to the server and the handshake protocol of SSL/TLS can be performed. Notice that the server can verify the authenticity of the certificate even if the values of the hidden extensions are not revealed. Then, the server requires to see the value of the extensions of interest for the transaction and the client submits the values of the extensions in clear along with the random paddings. The server then verifies that the hashing of the value of the extensions concatenated with the random padding agree with what appears on the crypto

```
SERIALNUMBER
ID: NAME;VALIDITY
RES: hostname1:port1/path1
RES: hostname2:port2/path2
....
....
RES: hostnamen:portn/pathn
-----BEGIN CERTIFICATE-----
...
(a certificate encoded in PEM format)
...
-----END CERTIFICATE-----
.....
.....
-----BEGIN CERTIFICATE-----
...
(a certificate encoded in PEM format)
...
-----END CERTIFICATE-----
```

Figure 9: The structure of a message containing the list of masks.

certificate. Notice that the same idea can be applied to the fields that are not in the extension section; this can be useful when a digital certificate is sent on a public network in plain format.

We stress that crypto certificates are different from (and more flexible than) pseudonyms as in a crypto certificate the various fields of the certificate can be still disclosed in a verifiable way.

5 Implementation and Experimental Validation

In this section we describe our experimental work with SPSL. We start by discussing some implementation details and then we describe the SPSL API that is intended to be a general tool for the development of client/server applications that use SPSL. In Section 5.3, we describe a system for private and secure Web navigation based on SPSL.

5.1 Computing the *CertificateVerify* message

In SSL/TLS the *CertificateVerify* message contains the signature (verifiable using the client's certificate) of a *control message* M that is the concatenation of the MD5 digest and the SHA digest of the messages exchanged so far (for a total length of 36 bytes).

In SPSL the *CertificateVerify* message is defined in a slightly different way for STRONG and MEDIUM levels of anonymity. Suppose that the mask consists of l RSA keys, $(N_1, e_1), \cdots, (N_l, e_l)$, and that the client knows the private key (N_i, d_i) corresponding to the i-th public key. The field *proof_of_identity* of the *CertificateVerify* message contains l messages $m_1, \cdots, m_l$, one for each key in the mask, along with their signatures $s_1, \cdots, s_l$. The logical XOR of the 36 least significant bytes of messages $m_1, \cdots, m_l$ is equal to the control message M. The service manager verifies that s_j is a signature of m_j (and does so by using the public key (N_j, e_j)) and that logical XOR of the 36 least significant bytes of messages $m_1, \cdots, m_l$ is M.

We assume that all modules are longer than M: M is only 36 bytes long and RSA keys shorter than 64 bytes are not considered very secure. Following the basic identification protocol, the client picks a random $s_j < N_j$, for each $j \neq i$, and sets $m_j = s_j^{e_j} \bmod N_j$. Messages m_j, for $j \neq i$, and the control message M determine a 36-byte message $\hat{m}_i$. Message $\hat{m}_i$ is then augmented to m_i with as many random bits as necessary to reach the length of modulus N_i. Message m_i is the message to be signed. If $m_i < N_i$ then the client computes signature s_i of m_i. If instead $m_i \geq N_i$ then the client should not sign m_i and start again picking different random s_j's. To be precise, m_i can be signed using secret key (N_i, d_i) (for example, by signing the digest of m_i) but then m_i would be the only message longer than the modulus and the identity of the client would be revealed. Notice that the probability that the whole process has to be started again is at most 1/2. Indeed, m_i has the same length as N_i and the most significant bit of N_i is 1.

5.1.1 Reducing the computation time

Every time the client must send a proof to the server the value of the control message M may differ. However, observe that values s_j and m_j, for $j \neq i$, do not depend on M and can thus be precomputed. This has the effect of dramatically reducing the time needed to perform the protocol: once M is determined, the client has only to sign message m_i.

5.2 The SPSL API

In this section we present the API that we have designed and implemented for the SPSL protocol. As it is shown is Figure 10 and Figure 11, it is very simple to develop SPSL applications or to modify existing TLS applications to work with SPSL.

We use the same data structures as the OpenSSL library [23]. In particular, `STACK_OF(TYPE)` denotes a stack data structure each element being of type `TYPE`.

The actual SPSL handshake is started when the client calls the `SSL_connect` function and the server is waiting in the `SSL_accept`. Before the actual call, on both sides, it must be specified that an SPSL connection is sought (as opposed to a regular SSL/TLS connection) and what the mask is going to be for the connection. A client obtains an SPSL connection by invoking the `SSL_set_cipher_list` function and by specifying a set of cipher suites corresponding to SPSL (the cipher suites specified will be sent as part of the *ClientHello* message). Several aliases for the most used sets of cipher suites have been defined; for example, `sPRVC` is an alias for the set of all SPSL cipher suites with strong anonymity level and `smPRVC` includes also those with medium anonymity level.

The mask to be used for the connection is specified by the following functions:

```
void SSL_set_mask_list(
    SSL *s, STACK_OF(X509 *) *list,
    unsigned char *id, char *mask_path)

void SSL_CTX_set_mask_list(
    SSL_CTX *ctx, STACK_OF(X509 *) *list,
    unsigned char *id, char *mask_path)
```

that define the mask associated with an SSL connection *s* or an SSL context *ctx*, respectively, and determine the content of the *CertificateRequest* and *CertificateList* (or *EncryptedCertificateList* for the strong anonymity level) messages of the protocol.

Depending on whether the functions are executed by the client or by the server, different messages are sent as part of the SPSL handshake. Let us look first at the server side starting with the case in which *id* is not NULL and points to a sequence of 4 bytes. As a consequence of the invocation of this function, if the server sends a *CertificateRequest* message[2] the four bytes pointed to by *id* are copied into the field `ID` of the message. The field `mask_list` of the *CertificateRequest* message is empty. When later the server will have to perform the handshake, it has two ways of accessing the actual list of certificates: either the first argument *list* points to the list of certificates; or the certificates are to be found in the file with the identifier of the set as name in the directory pointed to by the last argument. On the other hand, if *id* is NULL and *list* is not NULL, then the certificates belonging to the sequence *list* are copied into the field `mask_list` of the *CertificateRequest* message sent by the server. Finally, if both *id* and *list* are NULL, then both the fields `mask_list` and `ID` of the *CertificateRequest* message sent by the server are empty. In this case it is up to the client to specify the mask to be used by the connection by means of a *CertificateList* message. In this last case, the client might specify the mask by giving the identifier of the mask. Then, the last argument of the function specifies the directory in which to look for the file containing the certificates for the mask whose identifier is sent by the client.

On the client side, the invocation of the function has different effects depending on whether the *CertificateRequest* message is empty (both the `mask_list` and `ID` fields are empty). If the *CertificateRequest* message is empty then the client is required to send a *CertificateList* message. The content of the *CertificateList* message is determined using the same rules as for the server with the only exception that the client is not allowed to send an empty *CertificateList*. If instead *CertificateRequest* message is non-empty then the message *CertificateList* is not sent by the client. In case the *CertificateRequest* message specifies the mask by means of an identifier, the last argument of the function is the directory in which to look for the file containing the certificates in the mask.

After the mask has been set by calling the functions above, but before the SPSL handshake has started as an effect of a call to `SSL_connect`, more certificates can be added to the current mask by invoking the functions

`int SSL_add_mask(SSL *`*s*`,X509 *`*cert*`)`

`int SSL_CTX_add_mask(SSL_CTX *`*ctx*`,X509 *`*x*`)`.

that add the certificate pointed to by *cert* to the current mask associated with the SSL connection *s* or with context *ctx*.

Moreover, for efficiency reasons, the client can choose to perform the handshake by considering only a subset of the current mask (see discussion in Section 5.3.3). The function `STACK_OF(X509 *) *SSL_set_nmask(SSL *`*s*`, int` *n*`, int *`*pos*`)` marks as useful for the SSL connection *s* only the certificates referenced to by the first *n* entries of the array *pos* (the certificate must be assigned to *s* with another function).

The list of certificates to be used as parameter for the functions that specify the mask can

[2] For some levels of access control no mask is required and thus no *CertificateRequest* is sent. See the discussion regarding the function that specifies the access control level required.

be constructed using the following functions.

1. `STACK_OF(X509 *) *SSL_load_mask_file(char *`*fn*`)`
   This function reads the X.509 certificates contained in file named *fn* and constructs a sequence of certificates that is returned. The certificates are assumed to contain an RSA public key and must be in PEM format.

2. `STACK_OF(X509 *) *SSL_load_x_mask_file(char *`*fn*`, int *`*pos*`, int` *nm*`)`
   As before, only the certificates whose position is specified by the array *pos* are returned. More precisely, the file *fn* contains a list of certificates and the array *pos* contains a list of integers. The integers in *pos* are the positions within the sequence of certificates contained in the file *fn* of the certificates to be read. The certificates must be in PEM format. The parameter *nm* specifies the length of the array *pos*.

Once an SPSL connection has been established, the actual mask associated with the connection is visible to the application via the following functions:

1. `STACK_OF(X509 *) *SSL_get_mask_list(SSL *`*s*`)`
   `STACK_OF(X509 *) *SSL_CTX_get_mask_list(SSL_CTX *`*ctx*`)`
   These functions return the mask associated with the SSL connection *s* or the SSL context *ctx*.
2. `unsigned char *SSL_get_mask_id(SSL *`*s*`)`
   `unsigned char *SSL_CTX_get_mask_id(SSL_CTX *`*ctx*`)`
   These functions return the identifier of the mask associated with the SSL connection *s* or the SSL context *ctx*.
3. `STACK_OF(X509 *) *SSL_get_peer_mask_list(SSL *`*s*`)`
   `unsigned char *SSL_get_peer_mask_id(SSL *`*s*`)`
   These functions return the mask or the identifier that the peer has requested.

Similarly to TLS, the server can specify different levels of access control by invoking
`void SSL_CTX_set_access_control(SSL_CTX *`*ctx*`, int` *md*`, int (*`*cb*`) (SSL *, unsigned char *, int))`
`void SSL_set_access_control(SSL *`*s*`, int` *md*`, int (*`*cb*`) () ).`
We now describe the possible values for *md*:

1. `SSL_AC_NONE` no access control is performed.

2. `SSL_AC_YES` access control is optional; that is, a *CertificateRequest* message is sent by the server. The client can respond with a *CertificateVerify* message (containing a proof) or ignore the request of the server.

3. `SSL_AC_FAIL_IF_NO_PEER_PROOF | SSL_AC_YES` access control is required. That is, as before, a *CertificateRequest* message is sent by the server. If the client refuses or fails to send a *CertificateVerify* message then the connection is aborted (an alert message is sent).

The callback *cb* is used to verify the correctness of the access of the client. This is to allow the server application to perform extra checks on whether to grant access or not. If callback *cb* is NULL then the default verification including the verifications of the basic authentication protocol is performed. Otherwise, the function pointed to by *cb* is executed with the following

parameters: a pointer to the SSL session, a pointer to an array of `unsigned char` that contains the message of the client and an integer, that is the length of the message.

Once the SPSL connection has been established, the client issues a request for a resource. At this point the server needs to check whether the mask associated with the connection is appropriate for the request; that is, if the mask is a subset of the set of qualified certificates for the resource. This is achieved by invoking
`int SSL_access_control_verify(SSL *`*s*`, STACK_OF(X509 *) *`*st*`)`.
The value *st* points to the list of qualified certificates and the function returns 1 if the client is a qualified user, 0 if the client is not a qualified user, and -1 an error has been found. It is expected that the server application grants access if the return value is 1, denies access if the return value is 0, and shuts down the connection if the return value is -1.

SSLeay_add_ssl_algorithms();	initialize algorithms
ctx=SSL_CTX_new(SSLv3_client_method());	create a secure context
ssl=SSL_new(ctx);	create a free and secure connection
SSL_use_certificate_file(ssl, certfile, type);	certificate to be used is in file filename
SSL_use_PrivateKey_file(ssl, keyfile, type);	private key of the certificate in use is in file filename
st=SSL_load_mask_file(filename);	load the mask
SSL_set_mask_list(ssl, st, NULL, "path");	set the mask for the connection
SSL_set_cipher_list(ssl, "sPRVC");	set the acceptable cipher suites
sock=socket(....);	open a socket
SSL_set_fd(ssl,sock);	assign a file descriptor
connect(sock,....);	try to establish the connection
SSL_connect(ssl);	do a secure and private connect
SSL_write(ssl, buf_out, len_out);	do a secure and private write
SSL_read(ssl, buf_in, len_in);	do a secure and private read
SSL_shutdown(ssl);	close a secure connection
SSL_free(ssl);	free memory
SSL_CTX_free(ctx);	free memory

Figure 10: The skeleton of an SPSL client application.

5.2.1 Developing SPSL client/server applications

The SPSL API can be used as a general tool to develop privacy preserving client/server applications. In Figure 10 and in Figure 11, we present the skeleton of a client/server application developed using the SPSL API. The reader familiar with the OpenSSL API [23] will certainly notice that very few additional calls are needed to add SPSL support to existing secure applications developed using the OpenSSL API.

Let us now briefly discuss the skeleton of a client SPSL application. First of all, we notice that the client has to specify the certificate to be used for the anonymous authentication. He does so by invoking `SSL_use_certificate_file` and `SSL_use_PrivateKey_file`. Then, the client sets the list of cipher suites desired (this will influence the content of the *ClientHello* message). Next, the client specifies the mask to be used. In the case found in Figure 10, the client loads a list of certificates from a file. This will be the mask used for the connection in

```
SSLeay_add_ssl_algorithms();                    initialize algorithms
ctx=SSL_CTX_new(SSLv3_server_method());         create a secure context
ssl=SSL_new(ctx);                               create a secure connection
SSL_use_certificate_file(ssl, certfile, type);  certificate to be used is
                                                in file filename
SSL_use_PrivateKey_file(ssl, keyfile, type);    private key of the certificate
SSL_set_cipher_list(ssl, smPRVC);               set the cipher suite
SSL_set_access_control(ssl, SSL_AC_YES, NULL);  set the type of access
                                                load the mask
SSL_set_mask_list(ssl, NULL, "A34B5599", path); set the mask for the connection
                                                the mask has ID=A34B5599
                                                and is found in file A34B5599
SSL_set_fd(ssl,sock);                           assign a file descriptor
SSL_accept(ssl);                                wait for incoming requests
SSL_read(ssl,buf,len);                          do a secure and private read
......
st=SSL_load_mask_file(file_rigth);              load the mask certificates
ret=SSL_access_control_verify(ssl, st);         verify the access for the client
if (ret<0)
    error(...);                                 an error occurred
    else if (ret==0)                            access must be denied
        SSL_write(ssl, "Forbidden ...", ...);
      else                                      access is granted
        SSL_write(ssl, "OK ...", ...);
SSL_shutdown(ssl);                              close a secure connection
SSL_free(ssl);                                  free memory
SSL_CTX_free(ctx);                              free memory
```

Figure 11: The skeleton of an SPSL server application.

case an empty *CertificateRequest* is received from the server. If instead the server specifies the mask by giving the identifier, then the client will search for the file containing the certificates that constitute the mask in the directory *mask_path* (see the call to `SSL_set_mask_list`). The call to `SSL_connect` then starts the actual SPSL handshake.

Also on the server side it is necessary to specify the list of desired cipher suites by invoking `SSL_set_cipher_list`; the cipher suite selected by the server among those specified by the client will then appear as part of the *ServerHello* message. The content of the *CertificateRequest* message is determined by the calls to `SSL_set_mask_list` and `SSL_set_access_control`. In the example found in Figure 11, the server chooses to perform access control (by passing `SSL_AC_YES` to `SSL_set_access_control`). Moreover, the server selects the mask by its id (A34B5599). These two calls have the effect that a *CertificateRequest* message with non-empty ID is sent during the handshake. Once the handshake is completed (and the `SSL_accept` returns) the server reads from the client the resource requested. In part of the code not shown in the figure, the server determines that the list of qualified certificates for the requested resource is found in a file whose name is in the variable `file_right`. It then invokes the function `SSL_access_control_verify` and, if necessary (*i.e.*, if the mask used to complete the handshake is not appropriate), the client is requested to perform a new handshake. Notice that this additional handshake is performed in a manner that is completely transparent to

the client application. Indeed, the `SSL_read` invoked by the client application to obtain the result of his request checks whether the message received from the server is a data message (in which case the data is decrypted and authenticated using the appropriate keys and passed to the application) or is a new non-empty *CertificateRequest* message (in which case a new *CertificateVerify* message is generated along with the necessary proof and sent).

5.3 A private navigation system

To test the viability of our proposal, we have used the SPSL library to develop an HTTP client and an HTTP server. All the software developed is open source and can be obtained from the SPSL Project web site [30]. We start by discussing the server side.

5.3.1 An SPSL capable web server

On the server side, we have chosen the Apache web server [1] as it is widely deployed and it can be easily extended via modules. In order to save development time, we have decided to add SPSL capabilities to the well-known ModSSL [21] module that implements SSL/TLS. The new module is called ModSPSL. ModSPSL offers to the web manager the possibility to configure access to the resources using the following two new directives:

1. `SPSLAccessFile`

   Syntax: `SPSLAccessFile` *path-file:ID*
   This directive specifies the path (*path-file*) of the file that holds the certificates corresponding to qualified users; the client must give a proof of identity based on this set (or on a subset) of certificates. The file that holds the certificates must be in PEM format. The *:ID* part of this directive is optional and specifies an identifier (*ID*) of eight hexadecimal characters (corresponding to four bytes) for the set.

   For example, the following directive within a directory context

   `SPSLAccessFile` */www/X509/mask.pem:031078F4*

   specifies that the resources in the directory are associated with the mask with id 031078F4. The certificates of the mask are found in the file with name */www/X509/mask.pem*.

2. `SPSLAccessPath`

   Syntax: `SPSLAccessPath` *path-dir*
   This directive specifies the path to a directory that holds groups of certificates corresponding to different access groups.

   For example, the following directive within global context

   `SPSLAccessPath` */www/X509/MASKS*

   specifies, in case the mask is given by the client by an ID, where to find the actual file containing the certificates.

5.3.2 The SPSL client proxy

The SPSL proxy application allows one to use the SPSL protocol from the Netscape Navigator browser. The proxy is a simple application that runs on the same machine as the browser. The

browser must be configured so that all requests are sent to the proxy that will then contact the server. If an SPSL server is contacted then the proxy will perform all the necessary steps as specified by the SPSL protocol, obtain the desired resource from the SPSL server and then pass it to the browser. For each connection, the proxy simulates an SSL/TLS certificate request from the server so that the browser prompts the user for the personal certificate to use for the connection. The proxy then extracts from the local browser database the private key corresponding to the selected personal certificate. The extracted private key is needed by the proxy to perform the SPSL authentication. If the local key database is encrypted, the proxy prompts the user for the passphrase used to encrypt the database. This is the only additional step required to the user in order to access an SPSL protected resource. Once the passphrase has been given to the proxy, the proxy operates so that SPSL is completely transparent to the user. Extracting the private key from the local database has been somehow complicated by the fact that the format of the database is not very well documented. Since other developers might find themselves in the need to extract a private key from the local database, we have developed a library, called `Moz2I`, and released it as open source software (see [22]).

Another important feature of the proxy is the handling of the new mime-type `application/x-x509-mask` messages. As explained in the previous sections, this mime type is used to identify messages containing a description of the masks associated with the various resources offered by a server (in this case, the various documents, CGI applications or servlets offered by an SPSL-capable HTTP server). Typically, whenever the user downloads such access lists the proxy receives from the web server a message with mime-type `application/x-x509-mask`. The message is not passed to the browser (which has no use for such a message) but instead the information contained in the body of the message is stored in its cache indexed by location. The proxy notifies the browser that the cache has been updated. Subsequently, whenever the browser issues a request for a resource the proxy searches its cache for an entry corresponding to the location (or a prefix of it). If a matching entry is found in the proxy cache, then the mask obtained is proposed by the proxy to the server if an empty *CertificateRequest* message is received from the server.

5.3.3 Performance evaluation

The main drawback of SPSL is that the client has to compute a number of encryptions that is linear in the number of qualified members. Quantifying how the computational effort requested by SPSL to the client and the server affects the viability of SPSL is a major point in the experimental evaluation of our prototype.

We start by making two observations. First of all, observe that most of the computation performed by the client and the server consists in computing RSA encryptions. Typically, RSA public keys have small exponents (in our implementation based on OpenSSL all RSA public keys have exponent $2^{16} - 1$) and thus encryption requires only a few multiplications.

A second observation is more heuristic in nature. Suppose that, for a mask of l public keys, user U sends messages and corresponding signatures for a subset of l' public keys and that the XOR of the message is equal to m. Then one can easily convince himself that the manager M should still accept the request of user U: U certainly belongs to the group of qualified users. On the other hand, U is now hiding his identity in a smaller set and is partially losing his anonymity. User U thus can pick his desired level of anonymity ranging from the maximum,

corresponding to $l' = l$, to the minimum, corresponding to $l' = 1$. The computational effort (and the amount of communication required) is proportional to the desired level l' of anonymity and can thus be picked according to the computational power of user U. We stress (and only discuss the matter briefly) that identification as a member of a subset can be dangerous for anonymity. For example, if a user always picks the same (fixed but arbitrarily chosen) set of l' public keys then his transactions can be linked and loss of anonymity in one of them for external factors can compromise them all. On the other hand, if each time the user picks a random subset, then the service manager can obtain information on the identity of the user by computing the intersection of the sets. Instead, we suggest the following strategy to be used in conjunction with mask lists. Given the value of $l' < l$ and assuming, for the sake of simplicity, that l' divides l, the user considers the mask list for the resource he is interested in as $\frac{l}{l'}$ blocks each of l' certificates. The user will then identify himself using the set of certificates corresponding to the one block that contains his own certificate. Thus the user will always use the same set of certificates (and so it is no use for the service manager to compute intersections of sets) and so will all the l' owners of the certificates in the same block (so that loss of privacy in one transaction will not affect other transactions).

The SPSL web navigation system performances have been measured by running several times the web server Apache with our SPSL module, the SPSL proxy and the Netscape Navigator browser. The experimental results gathered show that SPSL does not seriously affect the performances of the client and the server. In Table 1 the results of the experiments are presented; the cases differ by the size of the mask (100, 200, 1000, 10000, and 100 and 500 out of a set of 10000) and whether an identifier has been used to specify the set (the first row) or the actual set of certificates has been sent (the second row). When an identifier is used to specify the list of qualified users, the server sends a *CertificateRequest* message with an empty list of certificates and a NULL identifier. Then the client sends the corresponding identifier as part of the *EncryptedCertificate* message. We assume that, in a previous session, the SPSL proxy has obtained the list of qualified users and the identifier for the resource that is being sought.

Each cell reports the average over several runs of the number of seconds elapsed between the request of a page by the browser and the reply of the web server. The experiments have been performed on Pentium III-based PCs with 256 Mb of RAM, running Linux 2.4.10. The server and the client are connected through 100Mb Fast Ethernet LAN. Only the more interesting (and CPU intensive) case of STRONG anonymity level has been considered. As expected, the overhead of the protocol grows with the size of the qualified set (see the first 4 columns in Table 1). However, we observe that it might be sufficient for a user to hide his identity in a set of a few hundreds of users. Therefore, in case in which the set of qualified users is large (in the order of several thousands), the user can choose a random subset of the set of qualified users and give a proof only relatively to the selected subset (see field *subset* of message *CertificateList* in Section 3.4). Column 5 and 6 report the slowdown incurred into in the case in which the user picks a random set of 100 or 500 certificates out of 10000. The slowdown reduces dramatically and shows that SPSL is suitable even for large scale settings.

	100	200	1000	10000	100(10000)	500(10000)
ID	0.9	2.1	7.4	44.5	2.4	4.7
LIST	2.1	2.8	10.7	52.8	3.3	6.1

Table 1: Experimental results

6 Conclusions

The use of SPSL does not protect the user against an attack that tries to get information from the IP address from which the request originated. For this reason, we propose to use SPSL in conjunction with a system that hides network-level information [26, 19, 9, 25]. In this way we would get the benefit of untraceability and anonymity also in the context of subscription-based remote services.

It is obvious that SPSL needs to be adopted by the server, as it cannot operate without the server's collaboration. This is one of the greatest obstacle to a widespread use of SPSL as it is conceivable that servers will be very reluctant to allow anonymous identification or even the use of crypto certificates. It is thus important to make users aware of the threats to their privacy posed by current methods and that practical solutions are available. Identifying threats so to increase user awareness on this very sensitive aspects is one of the main motivations of this research.

Subscription is not the only form of payment for remote services. For example, the service manager might consider it more appropriate to charge on a per-access basis instead of charging a flat fee for a fixed period. We point-out that SPSL cannot be used in this context and that a different approach based on unduplicatable anonymous tokens (a concept similar to electronic cash) is necessary.

7 Acknowledgments

We thank Enzo Auletta, Claudio Chimera, Alfredo De Santis, Mimmo Parente, Mike Reiter and Moti Yung for several discussions related to the SPSL protocol.

We also thank Dr. Stephen Henson for his help in understanding the format of the Netscape private key database.

Finally, we would like to thank the anonymous reviewers for their careful reading of the paper and for their suggestions.

Part of this work has appeared in [24].

References

[1] The Apache web server project. http://www.apache.org, 2002.

[2] Giuseppe Ateniese, Jan Camenisch, Marc Joye, and Gene Tsudik. A practical and provably secure coalition-resistant group signature scheme. In Mihir Bellare, editor, *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, volume 1880 of *Lecture Notes in Computer Science*, pages 255–270. Springer, 2000.

[3] Giuseppe Ateniese and Gene Tsudik. Some open issues and new directions in group signatures. In M. Franklin, editor, *Financial Cryptography*, volume 1648 of *Lecture Notes in Computer Science*, pages 196–211. Springer Verlag, 1999.

[4] Stefan Brands. *Rethinking Public Key Infrastructures and Digital Certificates; Building in Privacy.* MIT Press, 2000.

[5] J. Camenisch. Efficient and generalized group signatures. In W. Fumy, editor, *Advances in Cryptology – Eurocrypt '97*, volume 1223 of *Lecture Notes in Computer Science*, pages 465–479. Springer Verlag, 1997.

[6] D. Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology – Crypto 82*, Lecture Notes in Computer Science, pages 199–203. Springer Verlag, 1983.

[7] D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, pages 1035–1044, 1985.

[8] D. Chaum and E. van Heyst. Group signatures. In D. W. Davies, editor, *Advances in Cryptology – Eurocrypt '91*, volume 547 of *Lecture Notes in Computer Science*, pages 257–265. Springer Verlag, 1991.

[9] David Chaum. Untraceable electronic mail, return addresses and digital pseudonyms. *Communication of the ACM*, 24(2):84–90, February 1981.

[10] Alfredo De Santis, Giovanni Di Crescenzo, and Pino Persiano. Communication-efficient anonymous group identification. In *Proc. of the 5th ACM Conference on Computer and Communications Security.* ACM, 1998.

[11] Alfredo De Santis, Giovanni Di Crescenzo, Pino Persiano, and Moti Yung. On monotone formula closure of SZK. In *Procedings of the 35th IEEE Symposium on Foundations of Computer Science (FOCS94)*, pages 454–465, 1994.

[12] T. Dierks and C. Allen. The TLS protocol version 1.0. Network Working Group RFC 2246, 1999.

[13] Olivier Dubuisson. *ASN.1 – Communication between heterogeneous systems.* Morgan Kaufmann Publisher, 2000.

[14] D. Eastlake and P. Jones. US Secure Hash Algorithm 1. RFC 1374, 2001.

[15] U. Feige, A. Fiat, and A. Shamir. Zero-knowledge proofs of identity. *Journal of Cryptology*, 1:77–94, 1988.

[16] Uri Feige and Adi Shamir. Witness indistinguishable and witness hiding protocols. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 416–426, 1990.

[17] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol version 3.0. Transport Layer Security Working Group, Internet Draft, 1996. Available at http://home.netscape.com/eng/ssl3.

[18] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. *SIAM Journal on Computing*, 18:186–208, 1989.

[19] C. Gulcu and G. Tsudik. Mixing e-mail with babel. In *Symposium on Network and Distributed System Security*, pages 2–16, 1996.

[20] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X509 public key infrastructure: Certificate and Certificate Revocation List (CRL) Profile. Network Working Group, RFC 3280, April 2002.

[21] The ModSSL home page. http://www.modssl.org.

[22] The Moz2I home page. http://www.security.unisa.it/spsl/moz2i.html, 2000.

[23] The OpenSSL home page. http://www.openssl.org.

[24] P. Persiano and I. Visconti. User privacy issues regarding certificates and the TLS protocol (the design and implementation of the SPSL protocol). In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 53–62. ACM, 2000.

[25] M. Reed, P. Syverson, and D. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communication Special Issue on Copyright and Privacy Protection*, 1998.

[26] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM Transaction on Information and System Security*, 1(1):66–92, 1998.

[27] R. Rivest. The MD5 message-digest algorithm. RFC 1321, 1992.

[28] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In C. Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 552–565. Springer Verlag, 2001.

[29] S. Schechter, T. Parnell, and A. Hartemink. Anonymous authentication of membership in dynamic groups. In Matthew Franklin, editor, *Proceedings of the Third International Conference on Financial Cryptography 99*, volume 1648 of *Lecture Notes in Computer Science*, pages 184–195. Springer Verlag, 1999.

[30] The SPSL home page. http://www.security.unisa.it/spsl.

[31] S. G. Stubblebine, P. F. Syverson, and D. M. Goldschlag. Unlinkable serial transactions: protocols and applications. *ACM Transactions on Information and System Security*, 2(4):354–389, 1999.

[32] W3C. Resource Description Framework (RDF) Model and Syntax Speficitation. REC-rdf-syntax-19990222, 1999.