This extended abstract appeared in
Proceedings of the 2006 ACM Symposium on Information, computer and communications security (AsiaCCS '06)
(March 21–24, 2006, Taipei, Taiwan), pages 35–45, ACM Press, New York, NY, USA.

Provably Secure Password-Based Authentication in TLS

Michel Abdalla[1], Emmanuel Bresson[2], Olivier Chevassut[3], Bodo Möller[4], and David Pointcheval[1]

[1] Departement d'Informatique, École normale supérieure
45 Rue d'Ulm,
75230 Paris Cedex 05, France
{Michel.Abdalla,David.Pointcheval}@ens.fr
[2] Cryptology department, CELAR Technology Center
35174 Bruz Cedex, France
Emmanuel.Bresson@m4x.org
[3] Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
OChevassut@lbl.gov
[4] Dept. of Mathematics and Statistics, University of Calgary
Calgary, AB T2N 1N4, Canada
bmoeller@acm.org

Abstract. In this paper, we show how to design an efficient, provably secure password-based authenticated key exchange mechanism specifically for the TLS (Transport Layer Security) protocol. The goal is to provide a technique that allows users to employ (short) passwords to securely identify themselves to servers. As our main contribution, we describe a new password-based technique for user authentication in TLS, called *Simple Open Key Exchange* (SOKE). Loosely speaking, the SOKE ciphersuites are unauthenticated Diffie-Hellman ciphersuites in which the client's Diffie-Hellman ephemeral public value is encrypted using a simple mask generation function. The mask is simply a constant value raised to the power of (a hash of) the password.
The SOKE ciphersuites, in advantage over previous password-based authentication ciphersuites for TLS, combine the following features. First, SOKE has formal security arguments; the proof of security based on the computational Diffie-Hellman assumption is in the random oracle model, and holds for concurrent executions and for arbitrarily large password dictionaries. Second, SOKE is computationally efficient; in particular, it only needs operations in a sufficiently large prime-order subgroup for its Diffie-Hellman computations (no safe primes). Third, SOKE provides good protocol flexibility because the user identity and password are only required once a SOKE ciphersuite has actually been negotiated, and after the server has sent a server identity.

Keywords. Encrypted Key Exchange, Password Authentication, TLS.

1 Introduction

1.1 Motivation

An increasing number of Internet systems all around the world are using open-source products. For examples, in the categories of operating systems, browsers, webservers, and cryptographic engines, open-source software products such as Linux, Mozilla, Apache, and OpenSSL are widely fielded. For building Grid systems and applications, the reference is the Globus toolkit[1]. This toolkit unlocks the computing power of the Internet by enabling users to tap a global network of computer systems to access information and large distributed processing power [16]. Grid security is based on the recognition that individual sites have their own local security policies and methods to enforce it [17]. Joining a grid does not force a site to radically change its security policies already in place, but rather to make them inter-operable with the general grid policy. This requirement led to the implementation in the Globus Toolkit of a security infrastructure based on the OpenSSL software and X.509 identity certificates. Recent security compromises of user and server machines, however, have resulted in site security policy changes. Many sites are changing their security policies to prohibit long-term private keys associated with X.509 certificates from being stored on a user's machine. The keys will instead be stored on servers in data centers where they can be protected better.

[1] http://www.globus.org/

The immediate need for Grids is a cryptographic technology that allows users to securely identify themselves to data centers using passwords (short enough to be memorized) and be issued short-lived public/private keys [15]. The simplest approach is to use the OpenSSL implementation of the TLS protocol's unilateral-authentication mode [13] and add a straightfoward password check: the (credential repository) server sends its X.509 certificate for explicit verification, then the user sends his password through the TLS secure channel. This is what typically happens in the MyProxy (open-source) software for managing grid security credentials[2], and generally in most of today's "secure" WWW applications. This approach, however, provides a false sense of security because the privacy of the user's password is lost if the server is actually not authentic ("phishing" attacks), and, hence, anything protected through the password could get compromised. The password can be exposed when a user accepts a bogus server certificate, which usually takes just a single mouse click. And this is not all—security is in fact totally dependent on the X.509 public-key infrastructure used for server authentication: any party with a certificate apparently issued to the legitimate server can potentially obtain the password from the client. A single trusted Certification Authority (CA) that is malicious or careless would do lot of harm! Sending a one-time password in place of the fixed password would not help either since the bogus server could turn around and use this one-time password to impersonate the user.

1.2 Contributions

This paper alleviates the above problem by presenting a technique that ties the user's authentication to the TLS secure channel. We present a high-level description of this technique in TLS and a security analysis in the tradition of provable security. The technique goes under the name of *Simple Open Key Exchange* (SOKE). Using SOKE for TLS combines the following three features, in advantage over previous password-based authentication ciphersuites in TLS [23, 24]:

1. SOKE has formal security arguments (in the random oracle model under the computational Diffie-Hellman (CDH) assumption, based on Abdalla *et al.*'s security definitions for password-based authenticated key exchange [3]).
2. SOKE is computationally efficient because it can work in an appropriate subgroup of the multiplicative group of a prime field without requiring a "safe prime"; similarly, it could be used with elliptic curves or other groups. Thus, the computations for SOKE can use smaller exponents than those required for [23] or [24]. (The protocol described in [23] does not need safe primes, but it does not solely work in the subgroup. Rather, it involves an exponentiation to map arbitrary elements of $\mathbf{Z}_p^*$ to subgroup elements: the smaller the subgroup, the larger the exponent for this exponentiation. SOKE, in contrast, uses *only* the subgroup.)
3. SOKE does not require conveying the user identity in the very first TLS handshake messages (known as `ClientHello`). This feature (also present in [23], but not in [24]) provides additional protocol flexibility: the user does not have to specify a user identity and password before a SOKE ciphersuite has actually been negotiated, so these values can be chosen depending on the server identity transmitted by the server.

SOKE ciphersuites for TLS (SOKE-TLS in short) are essentially unauthenticated Diffie-Hellman ciphersuites where the client's Diffie-Hellman ephemeral public value is encrypted under the password shared with the server; the encryption primitive is a mask generation function computed as the product of the message with a constant value raised to the power of the password. Full TLS handshakes negotiating a SOKE ciphersuite require modifications to the usual TLS handshake message flow to achieve security: the usual `Finished` messages of the TLS protocol, which are sent under the newly negotiated cryptographic parameters (after `ChangeCipherSpec`), are replaced by `Authenticator` messages, which

[2] http://grid.ncsa.uiuc.edu/myproxy/

are similar in meaning to the `Finished` messages but must be sent under the old cryptographic parameters (namely, before `ChangeCipherSpec`)—that is, typically (in the case of an initial handshake) unencrypted. Also, while usually the client sends its `Finished` message first, here the server has to send its `Authenticator` message first. The client can only send its `Authenticator` message after having verified the server's `Authenticator` message (to avoid dictionary attacks since otherwise a rogue server could try a brute force attack on the client's password [23]).

1.3 Related Work

A Password-based Authenticated Key Exchange (PAKE) is a key exchange [14, 22] with one [19] or two flows [8, 9] encrypted using the password as a common symmetric key. Bellare *et al.* [5, 7], Boyko *et al.* [10], and MacKenzie [10, 20] proposed and proved secure various PAKE structures. These structures were later proved forward-secure under various computational assumptions [2, 11, 12, 18]. Instantiations for the encryption primitive were either a password-keyed symmetric cipher or a mask generation function computed as the product of the message with the hash of a password, until Abdalla *et al.* proposed the Simple PAKE (SPAKE-)structure with a new mask generation function computed as the product of the message with a constant value raised to the power of the password [4]. Whereas earlier mask generation functions need a full-domain hash function into the group, SPAKE provides high flexibility in the choice of groups (e.g., this makes it easy to work with elliptic curves). The SOKE structure, described in the present paper, is the SPAKE structure with only one flow encrypted.

1.4 Organization

The paper is organized as follows. In Section 2, we define the formal security model that we use through the rest of the paper. In Section 3, we present the algorithmic assumptions upon which the security of the SOKE scheme and, thus, SOKE-TLS is based. In Section 4, we describe SOKE-TLS itself, and in Section 5 we prove that it is secure via a reduction from SOKE-TLS to the computational Diffie-Hellman problem. We finally conclude the paper by presenting our long-term objective and by describing the steps we have undertaken to achieve it.

2 Formal Model for Password Authentication

In this section, we recall the security model of Abdalla *et al.* [3] for password-based authenticated key exchange protocol, which in turn implies that of Bellare *et al.* [5].

2.1 Using Weak Passwords

A password-based authenticated key exchange protocol P is a protocol between two parties, a *client* $C \in \mathsf{client}$ and a *server* $S \in \mathsf{server}$. Each participant in a protocol may have several *instances*, called oracles, involved in distinct, possibly concurrent, executions of P. We let U^i denote the instance i of a participant U, which is either a client or a server.

Each client $C \in \mathsf{client}$ holds a password pw_C. (Actually, in practice a client implementation will typically work on behalf of a *user,* where a single user may be using the same password with different client implementations, and where multiple users may be employing the same client implementation that queries the current user for the respective password. Also, a single user may be acting as multiple virtual users by using different passwords in different contexts. However, in the abstract view of the security model, we neglect these fine points and consider only abstract clients instead, which combine the knowledge of a single password with the computational capabilities of a client implementation.)

If a server $S \in \mathsf{server}$ has been set up for key exchange with a given client C, this means that the respective server was handed a value $pw_S[C]$ for said client. The value $pw_S[C]$ is denoted a *derived*

password [5]. Schemes where $pw_S[C] = pw_C$ are called symmetric; in general, $pw_S[C]$ may differ from pw_C.

pw_C and $pw_S[C]$ are also referred to as the *long-lived keys* of client C and server S. Each password pw_C is considered to be a low-entropy string, drawn from the dictionary $\mathsf{Password}$ of size N, according to the uniform distribution.

2.2 Security Model

The interaction between an adversary $\mathcal{A}$ and the protocol participants occurs only via oracle queries, which model the adversary capabilities in a real attack (see literature for more details [5, 12, 3].) The types of oracles available to the adversary are as follows:

- $\mathsf{Execute}(C^i, S^j)$: The output of this query consists of the messages exchanged during the honest execution of the protocol. This models passive attacks.
- $\mathsf{Send}(U^i, m)$: The output of this query is the message that the instance U^i would generate upon receipt of message m. A query $\mathsf{Send}(U^i, \text{“start”})$ initializes the key exchange protocol, and thus the adversary receives the initial flow that the initiator would send to the receiver. This models active attacks.

2.3 Security Notions

In order to define a notion of security for the key exchange protocol, we consider a game in which the protocol P is executed in the presence of the adversary $\mathcal{A}$. In this game, we first draw a password pw from $\mathsf{Password}$, provide coin tosses and oracles to $\mathcal{A}$, and then run the adversary, letting it ask any number of queries as described above, in any order.

AKE Security. In order to model the privacy (semantic security) of the session key, we consider the game $\mathbf{Game}^{\mathsf{ake}}(\mathcal{A}, P)$, in which an additional oracle is made available to the adversary: the $\mathsf{Test}(U^i)$ oracle.

- $\mathsf{Test}(U^i)$: This query tries to capture the adversary's ability to distinguish real keys from random ones. In order to answer it, we need a private random coin b (unique for the whole game) and then forward to the adversary either the session key sk held by U^i if $b = 1$ or a random key of the same size if $b = 0$.

The Test-oracle can be queried as many times as the adversary $\mathcal{A}$ wants, but is available only if the queried instance U^i holds a key, i.e. has accepted the key exchange session. The simple restriction is that in the random case the same random answer will be obtained when querying two partners. (Two player instances are said to be partners if they have exchanged the same set of messages, and are thus expected to hold the same session key.) When playing this game, the goal of the adversary is to guess the hidden bit b involved in the Test-queries, by outputting a guess b'. Let Succ denote the event in which the adversary is successful and correctly guesses the value of b. The **AKE advantage** of an adversary $\mathcal{A}$ is then defined as $\mathsf{Adv}^{\mathsf{ake}}_P(\mathcal{A}) = 2\Pr[\mathsf{Succ}] - 1$. The protocol P is said to be (t, ε)-**AKE-secure** if $\mathcal{A}$'s advantage is smaller than ε for any adversary $\mathcal{A}$ running with time t. Note that the advantage of an adversary that simply guesses the bit b is 0 in the above definition due to the rescaling of the probabilities.

It is worth pointing out that, as proven in [3], any scheme that is proven secure in the above model is also secure in the model of Bellare *et al.* [5]. The converse, however, is not necessarily true due to the non-tightness of the security reduction (see [3] for more details).

Mutual Authentication. The above property essentially means that only the legitimate participants can obtain the secret session key, and that an outside adversary should not be able to learn information about the key. This is also known as *implicit authentication*. In addition to that property, a protocol is said to achieve *mutual authentication* if each party can be ensured that it has established a key with the players it intended to. In the context of password-based protocols, authentication between the players is often done through *authenticators*. Intuitively, an authenticator is a value that can only be computed (except with small probability) with the knowledge of a secret. The idea is that if a party has sent data in the clear, it must subsequently provide an authenticator relative to these data. We denote by $\mathsf{Succ}^{\mathsf{auth}_S}(\mathcal{A})$ the success probability of an adversary trying to impersonate the server (i.e., the probability that a client will finish the key exchange protocol accepting the adversary as an authenticated server).

Forward-Secrecy One additional security property to consider is that of forward-secrecy. A key exchange protocol is called forward-secure if the security of a session key between two participants is preserved even if one of these participants later is compromised. In order to consider forward-secrecy, one has to account for a new type of query, the Corrupt-query, which models the compromise of a participant by the adversary. This query is defined as follows:

– $\mathsf{Corrupt}(U)$: This query returns to the adversary the long-lived key pw_U for participant U. As in [5], we assume the weak corruption model in which the internal states of all instances of that user are not returned to the adversary.

In order to define the AKE success probability in the presence of this new type of query, one should extend the restriction in the random case: all the executions completed after the Corrupt-query are answered with the real session key.

Let Succ denote the event in which the adversary successfully guesses the hidden bit b used by Test oracle. The **FS-AKE advantage** of an adversary $\mathcal{A}$ is then defined as $\mathsf{Adv}_P^{\mathsf{ake-fs}}(\mathcal{A}) = 2\Pr[\mathsf{Succ}] - 1$. The protocol P is said to be (t, ε)-**FS-AKE-secure** if $\mathcal{A}$'s advantage is smaller than ε for any adversary $\mathcal{A}$ running with time t.

3 Intractability Assumptions

3.1 Diffie-Hellman Problems

The arithmetic is in a finite cyclic group $\mathbb{G} = \langle g \rangle$ of order a ℓ-bit prime number q, where the operation is denoted multiplicatively.

$\mathsf{CDH}_{g,\mathbb{G}}$: Computational Diffie-Hellman. A (t, ε)-$\mathsf{CDH}_{g,\mathbb{G}}$ attacker, in a finite cyclic group $\mathbb{G}$ of prime order q with g as a generator, is a probabilistic machine Δ running in time t such that its success probability $\mathsf{Succ}^{\mathsf{cdh}}_{g,\mathbb{G}}(\Delta)$, given random elements g^x and g^y to output g^{xy}, is greater than ε:

$$\mathsf{Succ}^{\mathsf{cdh}}_{g,\mathbb{G}}(\Delta) = \Pr[\Delta(g^x, g^y) = g^{xy}] \geq \varepsilon.$$

We denote by $\mathsf{Succ}^{\mathsf{cdh}}_{g,\mathbb{G}}(t)$ the maximal success probability over every adversaries running within time t. The CDH-Assumption states that $\mathsf{Succ}^{\mathsf{cdh}}_{g,\mathbb{G}}(t) \leq \varepsilon$ for any t/ε not too large.

$\mathsf{PCCDH}_{g,\mathbb{G}}$: Password-Based Chosen-Basis Computational Diffie-Hellman. The so-called password-based chosen-basis computational Diffie-Hellman problem is a variation of the computational Diffie-Hellman that is more appropriate to the password-based setting: Let $\mathcal{D} = \{1, \ldots, n\}$ be a dictionary containing n equally likely password values and let $\mathcal{P}$ be a public injective map $\mathcal{P}$ from $\{1, \ldots, n\}$ into $\mathbb{G}$. Now let us consider an adversary that runs in two stages. In the first stage, the adversary is given as

input two random elements U and X in $\mathbb{G}$ as well as the public injective map $\mathcal{P}$ and it outputs a value Y in $\mathbb{G}$ (the chosen-basis). Next, we choose a random password $k \in \{1, \ldots, n\}$ and give it to the adversary. We also compute the mapping $r = \mathcal{P}(k)$ of the password k. The goal of the adversary in this second stage is to output the value $K = \mathsf{CDH}_{g,\mathbb{G}}(X, Y/U^r)$.

Note that an adversary that correctly guesses the password k in its first stage can easily solve this problem by computing $r = \mathcal{P}(k)$ and making, for instance, $Y = g \cdot U^r$ so that $K = X$. Since we assume k to be chosen uniformly at random from the dictionary $\{1, \ldots, n\}$, an adversary that chooses to guess the password and follow this strategy can succeed with probability $1/n$.

The idea behind the password-based chosen-basis computational Diffie-Hellman assumption is that no adversary can do much better than the adversary described above: that is, $\mathsf{Succ}^{\mathsf{pccdh}}(t, n)$ cannot be significantly larger than $1/n$, where n is the size of the dictionary.

$\mathsf{SPCCDH}_{g,\mathbb{G}}$: Set Password-Based Chosen-Basis Computational Diffie-Hellman. This is a generalization of the above problem, in which the adversary, in the second stage, outputs a set of candidates for K. It wins if the set indeed contains K: $\mathsf{Succ}^{\mathsf{spccdh}}(t, s, n)$ is the maximal probability to output a valid set of size s, within time t. The $\mathsf{SPCCDH}_{g,\mathbb{G}}$ problem is equivalent to the $\mathsf{CDH}_{g,\mathbb{G}}$ problem as shown in Appendix A.

3.2 Pseudo-Random Functions

A pseudo-random function family (PRF) is a family of functions $\mathcal{F} = (f_k)_k$ in $\mathcal{F}_{m,n}$, the set of the functions from $\{0,1\}^m$ into $\{0,1\}^n$, indexed by a key $k \in \{0,1\}^\ell$, so that for a randomly chosen ℓ-bit key k, no adversary can distinguish the function f_k from a truly random function in $\mathcal{F}_{m,n}$: we define the advantage $\mathsf{Adv}^{\mathsf{prf}}_{\mathcal{F}}(\mathcal{D}, q) = |\Pr_k[1 \leftarrow \mathcal{D}^{f_k}] - \Pr_f[1 \leftarrow \mathcal{D}^f]|$, where $\mathcal{D}$ is a distinguisher, with an oracle access to either a random instance f_k in the given family $\mathcal{F}$ or a truly random function f in $\mathcal{F}_{m,n}$, and must distinguish the two cases with at most q queries to the function oracle. We say that such a family is a (q, ε, t)-PRF if for any distinguisher asking at most q queries to the oracle, its advantage is less than ε, after a running time bounded by t.

3.3 Message Authentication Codes

A Message Authentication Code, say $\mathsf{MAC} = (\mathsf{MAC.Sign}, \mathsf{MAC.Verify})$, is made of the two following algorithms, with a secret key sk uniformly distributed in $\{0,1\}^\ell$:

- The *MAC generation algorithm* MAC.Sign. Given a message m and secret key $sk \in \{0,1\}^\ell$, MAC.Sign produces an authenticator μ. This algorithm might be probabilistic.
- The *MAC verification algorithm* MAC.Verify. Given an authenticator μ, a message m and a secret key sk, MAC.Verify tests whether μ has been produced using MAC.Sign on inputs m and sk.

The classical security level for MAC is to prevent existential forgeries, even for an adversary which has access to the generation and the verification oracles. We denote by $\mathsf{Succ}^{\mathsf{mac}}(t, q)$ the maximum success probability of a forger running within time t and making at most q queries to the MAC.Sign oracle.

4 A Simple Open Key Exchange for TLS (SOKE-TLS)

In this section, we describe the Diffie-Hellman Simple Open Key Exchange (SOKE), an efficient, provably secure password-based authenticated key exchange for TLS ciphersuites. The proof of security is based on the computational Diffie-Hellman assumption and holds in the random oracle model. The use of the random oracles, however, is very limited. More precisely, we only model a hash function as a random oracle during the extraction of the pre-master secret from the Diffie-Hellman result.

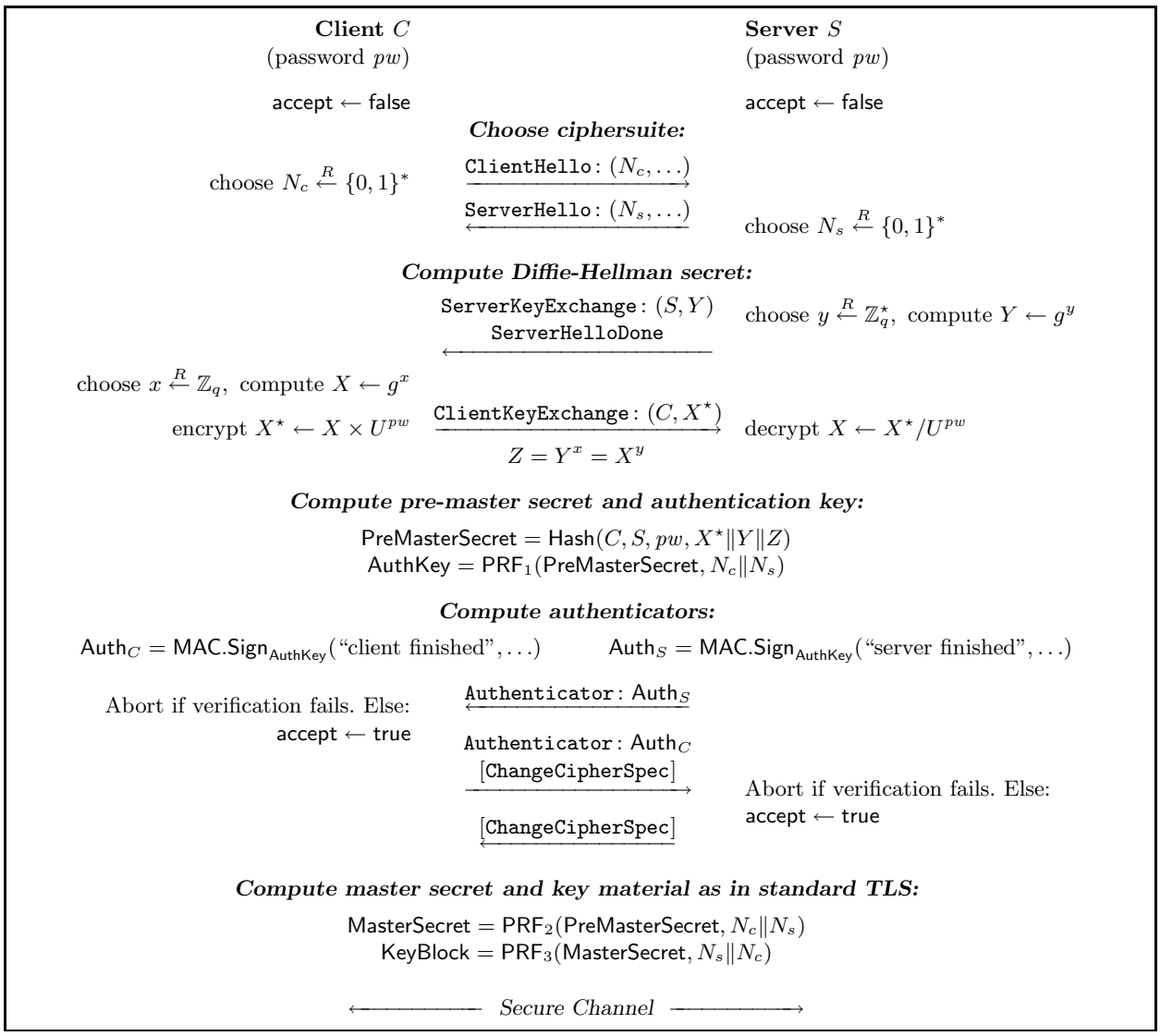

Fig. 1. The full handshake for SOKE-TLS ciphersuites.

As described below, the protocol does use the password in the key derivation process. As it will become clear in the next section, this simple enhancement plays a major role in the security of SOKE and enables us to prove its forward-security under concurrent executions without imposing any restrictions on the size of the dictionary.

4.1 The Handshake

Figure 1 illustrates the full TLS handshake for the case of SOKE-TLS ciphersuites. Since the abbreviated handshake for resuming a previously negotiated session is performed exactly as in other TLS ciphersuites, we only discuss the details of the full handshake.

1. **Choose Ciphersuite.** The client and the server negotiate the ciphersuite to use and exchange nonces.
   (a) The client sends its list of supported ciphersuites, including SOKE-TLS ciphersuites, in the TLS initial `ClientHello` message. It includes a nonce N_c as well. (In the TLS specification, this nonce is known as `ClientHello.random`.)

(b) The server specifies the ciphersuite to be used, selected from the client's list, by using a TLS `ServerHello` message. It also includes a nonce N_s (known as `ServerHello.random`).

Note. Each SOKE ciphersuite specifies a symmetric encryption algorithm (AES-CBC with either 128-bit or 256-bit keys) as well as an integrity algorithm (HMAC-SHA1) to use once the handshake has completed. Also, appriorate prime-order groups are standardized in the ciphersuite specification; they are equipped with two generators g and U that have been generated verifiably pseudo-randomly to provide assurance that no-one knows $\log_g U$ (see [1]).

2. **Compute Diffie-Hellman Secret.**
   (a) First, the server generates the random private exponent y and computes the Diffie-Hellman public key $Y = g^y$. The public key is sent to the client in the form of a `ServerKeyExchange` message, together with the server's identity.
   (b) If the server holds a private key and certificate suitable for signing, then in specific SOKE ciphersuites the server additionally sends the certificate to the client via a `Certificate` message. In this case the `ServerKeyExchange` message additionally contains a signature on the ephemeral Diffie-Hellman public key made with the server's long-term key—similar to non-anonymous standard Diffie-Hellman ciphersuites in TLS (cf. [13]).
   (c) The server then sends an empty message in the form of a `ServerHelloDone` to indicate that the "hello" phase is completed.
   (d) If the server has sent a `Certificate` message, the client verifies the signature.
   (e) The client generates its random private exponent x and computes the second Diffie-Hellman public key as $X = g^x$. That latter key is encrypted (using a password) as the product of the key with the password-based mask: $X^\star = g^x \times U^{pw}$. The client encapsulated its name and the encrypted value in the `ClientKeyExchange` message which is sent out.
   (f) The client and the server can now compute the common Diffie-Hellman secret $Z = g^{xy}$ from the values they received in the `ServerKeyExchange` and `ClientKeyExchange` messages, respectively.

Note. In the protocol description, C and S are strings giving the client identity and server identity, respectively (i.e., a user name and a server or "realm" name). The password for the user denoted by C is a string pw. Observe that the client identity C and the password pw are not used within the protocol before the server has chosen a SOKE ciphersuite and transmitted the server identity S. This means that a user only needs to provide C and pw after seeing that a SOKE handshake with S is going on. For example, HTTP servers might required password-based ciphersuites only for specific subtrees of their URL space by requesting a TLS renegotiation if necessary, and the server identity might depend on the specific URL.

A detail not shown in the figure is that the server, with its `ServerKeyExchange` message, will also send an index into the list of standardized groups for SOKE (such as $0, 1, 2, \ldots$ for standardized 1024-bit, 1536-bit, 2048-bit, ... prime moduli with appropriate subgroups). The client should display the server's choice of group to the client, and the user should provide his password only if he agrees with the group.

Note that when the client receives Y from the server and when the server receives $X^\star$ from the client, it is implicit that the respective recipient performs a group membership test: it is a fatal TLS handshake failure if Y or $X^\star$ is not a group member.

3. **Compute Pre-Master Secret and Authentication Key.**
   (a) The parties extract the randomness in the Diffie-Hellman result to form the pre-master secret as: $\mathsf{PreMasterSecret} = \mathsf{Hash}(C, S, pw, X^\star \| Y \| Z)$.
   (b) The pre-master secret is used as a means to derive the authentication key AuthKey used by the parties to perform the mutual authentication; we define $\mathsf{AuthKey} = \mathsf{PRF}_1(\mathsf{PreMasterSecret},$

$N_c \| N_s$). This key derivation is performed based on the standard TLS pseudo-random function PRF (see [13, Sec. 5]). The key derivation function $\mathsf{PRF}_1(\mathsf{PreMasterSecret}, z)$ used here is specific to SOKE ciphersuites; its value is obtained as $\mathsf{PRF}(\mathsf{PreMasterSecret}, \textit{"authentication key"}, z)$.

Note. Here the randomness extractor function $\mathsf{Hash}(z_1, z_2, z_3, z_4)$ is defined as a function with multiple inputs. The reason for this is that while we assume that group elements can be represented as fixed-length strings, the same does not hold for the strings C, S, and pw. A function $\mathsf{Hash}_1(z)$ in a single input can be used to implement Hash by defining $\mathsf{Hash}(z_1, z_2, z_3, z_4) = \mathsf{Hash}_1\big(\mathsf{SHA1}(z_1) \| \mathsf{SHA1}(z_2) \| \mathsf{SHA1}(z_3) \| z_4\big)$. The function $\mathsf{Hash}_1(z)$, in turn, can be instantiated by using $\mathsf{SHA1}(\text{constant}\langle 0\rangle \| z) \| \mathsf{SHA1}(\text{constant}\langle 1\rangle \| z) \| \ldots$ Other ways to instantiate Hash_1 are discussed in [6].

4. **Compute Authenticators.**
   (a) Both parties use AuthKey to produce the authenticators Auth_C and Auth_S. More precisely, the authenticator is set as a MAC on "*finished_label* ‖ *hash of handshake*", in which *finished_label* is the string "client finished" or "server finished", depending on which party is sending the respective message, and where *hash of handshake* denotes the hash of the concatenation of all the handshake messages sent so far in both directions exactly as would be used for the `Finished` message in other TLS ciphersuites (i.e., the MD5 hash concatenated with the SHA-1 hash). The server sends its `Authenticator` message first.
   (b) The client first checks the server's authenticator, and, if correct, sends its own `Authenticator`, and then proceeds to the `ChangeCipherSpec` message.
   (c) The server then checks the client's authenticator, and, if correct, replies with the message `ChangeCipherSpec` as well.

   Note. Usual TLS ciphersuites send `Finished` messages for authentication *after* switching to the newly negotiated key material (KeyBlock) in the TLS record layer (which event is indicated by a `ChangeCipherSpec` message). This approach, however, would violate the security notion that SOKE ciphersuites are designed to achieve. Instead, SOKE ciphersuites make use of `Authenticator` messages; the client does not send its `Authenticator` and `ChangeCipherSpec` before it has verified the server's `Authenticator`, and the server delays its `ChangeCipherSpec` until it has verified the client's `Authenticator`. Note that the `Authenticator` message does not undergo any processing using the KeyBlock, since it precedes the `ChangeCipherSpec`; this is different from the handshake in other TLS ciphersuites where `Finished` is sent in an TLS record processed under key material KeyBlock (see [13]).

5. **Compute Master Secret and Key Material.** The material KeyBlock is the bit string assigned to the Initialization Vectors (IVs), MAC secrets, and encryption keys which will protect the application sensitive messages. KeyBlock, exactly as in other TLS ciphersuites, is obtained indirectly, in two steps:
   (a) Both parties compute a common MasterSecret, using a function $\mathsf{PRF}_2(\mathsf{PreMasterSecret}, z)$ that is defined as $\mathsf{PRF}(\mathsf{PreMasterSecret}, \text{"mastersecret"}, z)$.
   (b) The MasterSecret is used to obtain KeyBlock as a function $\mathsf{PRF}_3(\mathsf{MasterSecret}, z)$, which is defined as $\mathsf{PRF}(\mathsf{MasterSecret}, \text{"key expansion"}, z)$. This two-stage derivation process is used by TLS session resumption: a new connection with new client and server nonces N_c and N_s can continue to use a previously negotiated MasterSecret and derive a new KeyBlock.

   Note. Another change from the standard TLS handshake message flow, besides having `Authenticator` sent before `ChangeCipherSpec`, is that for SOKE the server and not the client provides its authentication message first. This is necessary to protect clients against dictionary attacks: if the client was to make the start, its `Authenticator` message could be used by a malicious server that does not know pw to try out different passwords in an offline attack, looking which one results in the `Authenticator` message as observed.

4.2 Additional Remarks

How the Password Becomes an Exponent. Since the password pw appears as an exponent in the computations for SOKE ciphersuites, some additional hash is needed to obtain this exponent from the password string *password*. In the protocol description, we do not care about details of the hash and simply use the hash result pw (in the exponent space) as the "effective password" instead: anyone knowing pw is actually able to impersonate the client or the server, and the security proof shows that attacking the protocol reduces to finding pw. In other words, at the protocol level, pw is the password needed for authentication and *password* is just a way to remember it.

Password Derivation for Different Groups. Remember that the protocol should allow using one out of a set of different groups (the client likely wants to see the group being used before typing his password). So for group i, the effective password pw may be defined in the form

$$pw_i = \mathsf{hash}(i \parallel password) \bmod q_i.$$

Using the Password with Several Servers. The password string *password* should be hashed before being used as an exponent pw. If the protocol is changed again to use (C, S, pw) instead of just *password* to derive the exponent, we obtain a security improvement in practice in that a client can use the same *password* (string) with multiple servers and yet the respective secrets (effective passwords) will be different. (However, every server could run a dictionary attack to recover the common underlying password.)

5 Security Analysis of SOKE-TLS

Theorem 1. [FS-AKE Security] *Let us consider the above protocol, over a group of prime order q, where* Password *is a dictionary of size N, equipped with the uniform distribution. Let $\mathcal{A}$ be an adversary running within a time bound t that makes less than q_{active} active sessions with the parties and q_{passive} passive eavesdropping queries, and asks q_{hash} hash queries. Then we have*

$$\begin{aligned}\mathsf{Succ}^{\mathsf{auth}_S}_{\mathsf{SOKE}}(\mathcal{A}) \le{}& \frac{2q_{\mathsf{active}}}{N}\\ &+q_{\mathsf{hash}}(q_{\mathsf{active}}q_{\mathsf{hash}} + q_{\mathsf{active}} + 1) \times \mathsf{Succ}^{\mathsf{cdh}}(t + 2\tau_e)\\ &+2\frac{{q_{\mathsf{hash}}}^2}{2^{\ell_m}} + 2{q_{\mathsf{hash}}}^2 \times \mathsf{Succ}^{\mathsf{mac}}(t, 0) + \frac{{q_{\mathsf{hash}}}^2}{2^{\ell+1}}\\ &+2q_{\mathsf{session}} \times \mathsf{Adv}^{\mathsf{prf}}(t, 2) + \frac{{q_{\mathsf{active}}}^2}{2q} + \frac{{q_{\mathsf{passive}}}^2}{2q^2}\end{aligned}$$

$$\begin{aligned}\mathsf{Adv}^{\mathsf{ake-fs}}_{\mathsf{SOKE}}(\mathcal{A}) \le{}& \frac{6q_{\mathsf{active}}}{N}\\ &+2q_{\mathsf{hash}}(3q_{\mathsf{active}}q_{\mathsf{hash}} + q_{\mathsf{active}} + 1) \times \mathsf{Succ}^{\mathsf{cdh}}(t + 2\tau_e)\\ &+8\frac{{q_{\mathsf{hash}}}^2}{2^{\ell_m}} + 8{q_{\mathsf{hash}}}^2 \times \mathsf{Succ}^{\mathsf{mac}}(t, 0) + \frac{{q_{\mathsf{hash}}}^2}{2^{\ell}}\\ &+8q_{\mathsf{session}} \times \mathsf{Adv}^{\mathsf{prf}}(t, 2) + \frac{{q_{\mathsf{active}}}^2}{q} + \frac{{q_{\mathsf{passive}}}^2}{q^2}\end{aligned}$$

where τ_e denotes the computational time for an exponentiation in $\mathbb{G}$.

Proof. We are interested in the event S, which occurs if the adversary correctly guesses the bit b involved in the Test-queries. We furthermore consider server (unilateral) authentication: event A is set to true if a client instance accepts, without any server partner. Let us remember that in this attack game, the adversary is allowed to use Corrupt-queries.

Game $\mathbf{G}_0$: This is the real protocol, in the random-oracle model:

$$\begin{aligned} \mathsf{Adv}^{\mathsf{ake-fs}}_{\mathsf{SOKE}}(\mathcal{A}) &= 2\Pr[\mathsf{S}_0] - 1 \\ \mathsf{Adv}^{\mathsf{auth}_S}_{\mathsf{SOKE}}(\mathcal{A}) &= \Pr[\mathsf{A}_0]. \end{aligned} \tag{1}$$

Let us furthermore define the event $\mathsf{S}_{w/t}\mathsf{A} = \mathsf{S} \wedge \neg\mathsf{A}$, which means that the adversary wins the *Real-Or-Random* game without breaking authentication.

Game $\mathbf{G}_1$: In this game, we simulate the hash oracles (Hash, but also an additional hash function $\mathsf{Hash}' : (\{0,1\}^\star\times)^3 \to \{0,1\}^\ell$ that will appear in the Game $\mathbf{G}_3$ as usual by maintaining hash lists Λ_{Hash} and $\Lambda_{\mathsf{Hash}'}$. We also simulate all the instances, as the real players would do, for the Send-queries and for the Execute, Test and Corrupt-queries.

Game $\mathbf{G}_2$: We cancel games in which some collisions appear on the transcripts $(C, S, X^\star, Y)$, and on the master secrets. Regarding the transcripts, the distance follows from the birthday paradox since at least one element of each transcript is generated by an honest participant (at least one of them in each of the q_{active} active attacks, and all of them in the q_{passive} passive attacks). Likewise, in the case of the master keys, a similar bound applies since Hash is assumed to behave like a random oracle (which outputs ℓ-bit bit-strings):

$$\Pr[\mathsf{Coll}_2] \le \frac{{q_{\mathsf{active}}}^2}{2q} + \frac{{q_{\mathsf{passive}}}^2}{2q^2} + \frac{{q_{\mathsf{hash}}}^2}{2^{\ell+1}}. \tag{2}$$

Game $\mathbf{G}_3$: In this game, we show that the success probability of the adversary is negligible in passive attacks via Execute-queries. To do so, we modify the way in which we compute the pre-master secret PreMasterSecret in passive sessions that take place before or after a Corrupt-query. More precisely, whenever the adversary asks a Execute-query, we compute the pre-master secret PreMasterSecret as $\mathsf{Hash}'(C, S, X^\star\|Y)$ using the private oracle Hash' instead of the oracle Hash. As a result, it holds that any value of PreMasterSecret computed during a passive session becomes completely independent of Hash and $\mathsf{DH\text{-}Result}_{C/S}$, which are no longer needed in these sessions. Please note that the oracle Hash is still being used in active sessions.

The games $\mathbf{G}_3$ and $\mathbf{G}_2$ are indistinguishable unless the adversary $\mathcal{A}$ queries the hash function on $(C, S, pw, X^\star\|Y\|\mathsf{DH\text{-}Result}_{C/S})$, for such a passive transcript: we denote such event by AskH-Passive-Exe. In order to upper-bound the probability of this event, we consider an auxiliary game $\mathbf{G}_3$', in which the simulation of the players changes—but the distributions remain perfectly identical. Since we do not need to compute $\mathsf{DH\text{-}Result}_{C/S}$ for the simulation of Execute-queries, we can simulate Y as $V^{y^\star}$ and $X^\star$ as $g^{x^\star}$. If event AskH-Passive-Exe occurs, one can extract $K = \mathsf{CDH}_{g,\mathbb{G}}(V^{y^\star}, g^{x^\star}/U^{pw}) = V^{x^\star y^\star}/\mathsf{CDH}_{g,\mathbb{G}}(U,V)^{y^\star pw}$ from Λ_{Hash}:

$$\Pr[\mathsf{AskH\text{-}Passive\text{-}Exe}_3] \le q_{\mathsf{hash}} \times \mathsf{Succ}^{\mathsf{cdh}}_{g,\mathbb{G}}(t + 2\tau_e). \tag{3}$$

Game $\mathbf{G}_4$: In this game, we consider passive attacks via Send-queries, in which the adversary simply forwards the messages it receives from the oracle instances. More precisely, we replace Hash by Hash' when computing the value of PreMasterSecret whenever the values $(C, S, X^\star\|Y)$ were generated by oracle instances. Note that we can safely do so due to the absence of collisions in the transcript. Like in $\mathbf{G}_3$, any value PreMasterSecret computed during such passive sessions becomes completely independent of Hash and $\mathsf{DH\text{-}Result}_{C/S}$.

As in previous games, we can upper-bound the difference in the success probabilities of $\mathcal{A}$ in games $\mathbf{G}_4$ and $\mathbf{G}_3$ by upper-bounding the probability that $\mathcal{A}$ queries the hash function Hash on $(C, S, pw, X^\star\|Y\|$

$\mathsf{DH\text{-}Result}_{C/S}$), for such a passive transcript. We call this event $\mathsf{AskH\text{-}Passive\text{-}Send}$. Towards this goal, we consider an auxiliary game $\mathbf{G}_4$', in which the simulation of the players changes slightly without affecting the view of the adversary. In this simulation, we choose at random one of the $\mathsf{Send}(S, \text{“start”})$-queries being asked to S and we reply with $Y = V$ (hoping that is one of the sessions that the adversary simply forwards the message). On the client side, we change the simulation whenever it receives as input a message that was generated by a server instance, by computing $X^\star$ as $g^{x^\star}$. If the event $\mathsf{AskH\text{-}Passive\text{-}Send}$ occurs and our guess for the active session is correct, then we can extract $K = \mathsf{CDH}_{g,\mathbb{G}}(g^{x^\star}/U^{pw}, V) = V^{x^\star}/\mathsf{CDH}_{g,\mathbb{G}}(U,V)^{pw}$ from Λ_{Hash}:

$$\Pr[\mathsf{AskH\text{-}Passive\text{-}Send}_4] \leq$$
$$q_{\mathsf{active}} \times q_{\mathsf{hash}} \times \mathsf{Succ}^{\mathsf{cdh}}_{g,\mathbb{G}}(t + 2\tau_e). \quad (4)$$

Game $\mathbf{G}_5$: In this game, we make one of the most significant modifications. We replace the oracle Hash by the private oracle Hash' whenever the input to this query contains an element that was not generated by an oracle instance and no $\mathsf{Corrupt}$-query has occurred. More precisely, if either the value $X^\star$ or Y in the Hash-query $(C, S, pw, X^\star \| Y \| \mathsf{DH\text{-}Result}_{C/S})$ was generated by the adversary, then we reply to this query using $\mathsf{Hash}'(C, S, X^\star \| Y)$ as long as no $\mathsf{Corrupt}$-query has occurred. Clearly, the games $\mathbf{G}_5$ and $\mathbf{G}_4$ are indistinguishable as long as $\mathcal{A}$ does not query the hash function Hash on an input $(C, S, pw, X^\star \| Y \| \mathsf{DH\text{-}Result}_{C/S})$, where $\mathsf{DH\text{-}Result}_{C/S} = \mathsf{CDH}_{g,\mathbb{G}}(X^\star/U^{pw}, Y)$, for some execution transcript $(C, S, X^\star, Y)$. We denote this bad event by $\mathsf{AskHbC\text{-}Active}$. Thus,

$$|\Pr[\mathsf{A}_5] - \Pr[\mathsf{A}_4]| \leq \Pr[\mathsf{AskHbC\text{-}Active}] \quad (5)$$
$$|\Pr[\mathsf{S}w/t\mathsf{A}_5] - \Pr[\mathsf{S}w/t\mathsf{A}_4]| \leq \Pr[\mathsf{AskHbC\text{-}Active}]$$

Game $\mathbf{G}_6$: In this game, we replace the pseudo-random functions by truly random functions for all the sessions in which the value of $\mathsf{PreMasterSecret}$ has been derived with the private oracle Hash'. Since the value $\mathsf{PreMasterSecret}$ that is being used as the secret key for the pseudo-random function is independently and uniformly distributed, the distance can be proven by a classical sequence of hybrid games, where the counter is on the pre-master secrets. That is, each time a new pre-master secret is set, we increment the counter. Then, $\Pr[\mathsf{S}w/t\mathsf{A}_6] = \frac{1}{2}$.

$$|\Pr[\mathsf{A}_6] - \Pr[\mathsf{A}_5]| \leq q_{\mathsf{session}}\, \mathsf{Adv}^{\mathsf{prf}}(t, 1) \quad (6)$$
$$|\Pr[\mathsf{S}w/t\mathsf{A}_6] - \Pr[\mathsf{S}w/t\mathsf{A}_5]| \leq q_{\mathsf{session}}\, \mathsf{Adv}^{\mathsf{prf}}(t, 2).$$

Game $\mathbf{G}_7$: In this game, we exclude collisions on MAC keys for all the sessions in which the pre-master secret $\mathsf{PreMasterSecret}$ has been derived with the private oracle Hash'(which event is denoted $\mathsf{CollPRF}$). Let ℓ_m denote the length of MAC keys. Since for these sessions, the MAC keys are independently and uniformly distributed, the probabilities differ from those in the previous game by at most

$$\Pr[\mathsf{CollPRF}_7] \leq {q_{\mathsf{hash}}}^2/2^{\ell_m}. \quad (7)$$

Game $\mathbf{G}_8$: In this game, we exclude games wherein for some transcript $(C, S, X^\star, Y)$, there are two passwords pw_0 and pw_1 such that the corresponding pre-master secrets lead to a collision of the MAC-values (which event is denoted CollM).

Since we know that MAC-keys are truly random and different from each other at this point, the event CollM means that a MAC with a random key (one of the q_{hash} possible values) may be a forgery for another random key. Thus, by randomly choosing the two indices for the hash queries, we get the following upper-bound:

$$\Pr[\mathsf{CollM}_8] \leq {q_{\mathsf{hash}}}^2 \times \mathsf{Succ}^{\mathsf{mac}}(t, 0). \quad (8)$$

Game $\mathbf{G}_9$: Before proceeding with the analysis, we first split the event $\mathsf{AskHbC\text{-}Active}$ into two disjoint sub-cases depending on whether the adversary impersonates the client or the server. We denote these

events $\mathsf{AskHbC}_w\mathsf{S}$ and $\mathsf{AskHbC}_w\mathsf{C}$, respectively. In this game, we focus on $\mathsf{AskHbC}_w\mathsf{C}$ only. We now reject all the authenticators sent by the adversary for all the sessions in which the pre-master secret PreMasterSecret has been derived with the private oracle Hash': $\Pr[\mathsf{A}_9] = 0$. In order to evaluate the distance between the games $\mathbf{G}_9$ and $\mathbf{G}_8$, we consider the probability of the event $\mathsf{AskHbC}_w\mathsf{C}$, in which the adversary succeeds in faking the server by sending a valid authenticator before a Corrupt-query.

To evaluate the probability of event $\mathsf{AskHbC}_w\mathsf{C}$, we note that, up to the moment in which a Corrupt-query occurs, no information on the password pw of a user is revealed to the adversary, despite the fact that the password is still used in the computation of $X^\star$. To see that, note that, for any given transcript $(X^\star, Y)$ in which $X^\star$ was created by an oracle instance and for each password pw, there exists a value $x \in \mathbb{Z}_q$ such that $X^\star = g^x U^{pw}$ which is never revealed to the adversary. Moreover, since we have removed collisions on the pre-master secrets, on the MAC keys, and on the MAC values, there is at most one password that can lead to a valid authenticator. As a result, the probability that the adversary succeeds in sending a valid authenticator in each session is at most $1/N$. Thus, we get

$$\Pr[\mathsf{AskHbC}_w\mathsf{C}_9] \leq q_{\mathsf{fake-server}}/N. \tag{9}$$

Game $\mathbf{G}_{10}$: We finally concentrate on the success probability of the adversary in faking the client. What we show in this game is that the adversary cannot eliminate more than one password in the dictionary by impersonating a client. To do so, we first upper-bound the probability that, for some transcript $(X^\star, Y)$ in which Y was created by server instance, there are two hash queries in Λ_{Hash} such that one has $(X^\star, Y, pw_0, K_0 = \mathsf{CDH}_{g,\mathbb{G}}(X^\star/U^{pw_0}, Y))$ and $(X^\star, Y, pw_1, K_1 = \mathsf{CDH}_{g,\mathbb{G}}(X^\star/U^{pw_1}, Y))$. We denote this event CollH.

In order to upper-bound the probability of event CollH, we consider an auxiliary game in which the simulation of the players changes slightly without affecting the view of the adversary. The goal is to use the adversary to help us compute the computational Diffie-Hellman value of U and V. In this simulation, we choose at random one of the $\mathsf{Send}(S, \text{“start”})$-queries being asked to S and we reply with $Y = V$ in the hope that this is the session which leads to a collision in the transcript. For all other sessions, Y is simulated as g^y. Now, let us assume that the event CollH happens. If our guess for the $\mathsf{Send}(S, \text{“start”})$-query was correct, then we can extract the value $\mathsf{CDH}_{g,\mathbb{G}}(U, V)$ as $(K_1/K_0)^u$, where u is the inverse of $(pw_0 - pw_1)$, by simply guessing the indices of the two hash queries involved in the collision. We note that u is guaranteed to exist since $pw_0 \neq pw_1$. It follows that

$$\Pr[\mathsf{CollH}] \leq q_{\mathsf{send}} \times {q_{\mathsf{hash}}}^2 \times \mathsf{Succ}^{\mathsf{cdh}}_{g,\mathbb{G}}(t + \tau_e) \tag{10}$$

When the event CollH does not happen, for each transcript $(X^\star, Y)$ in which Y was created by server instance, there is at most one password value pw such that the tuple $(X^\star, Y, K = \mathsf{CDH}_{g,\mathbb{G}}(X^\star/U^{pw}, Y))$ is in Λ_{Hash}. As a result, the probability of the adversary in impersonating a client reduces to trying one password at a time. Thus,

$$\Pr[\mathsf{AskHbC}_w\mathsf{S}_{10}] \leq q_{\mathsf{fake-client}}/N + q_{\mathsf{send}} \times {q_{\mathsf{hash}}}^2 \times \mathsf{Succ}^{\mathsf{cdh}}_{g,\mathbb{G}}(t + \tau_e)$$

This concludes the proof of Theorem 1. □

6 Conclusion

The present paper describes *efficient* and *provably secure* methods for password-based authentication in the TLS protocol. Besides formal security arguments and good efficiency, these SOKE-TLS ciphersuites offer a message flow that is convenient in practice because the user identity and password need only be supplied if one of these ciphersuites has actually been negotiated in the TLS handshake. We provide

a low-level specification of these ciphersuites for TLS in [1]. In that document (intended for eventual publication as an Internet RFC), we specify the ciphersuites, provide standard group parameters and correct instantiations for the cryptographic transformations, define data structures for the handshake messages and the formatting of data on the wire, as well as additional handshake protocol details. The final goal of the authors is to publish as open source an implementation of these ciphersuites for the OpenSSL software toolkit conforming to this specification. This will, we hope, benefit open-source products such as the Globus Toolkit whose communications' security is based on the OpenSSL software.

7 Acknowledgments

The first and fifth authors have been supported in part by the European Commission through the IST Program under Contract IST-2002-507932 ECRYPT. The third author was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, Mathematical Information and Computing Sciences Division, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. This document is report LBNL-57609. See `http://www-library.lbl.gov/disclaimer`.

References

1. Michel Abdalla, Emmanuel Bresson, Olivier Chevassut, Abdelilah Essiari, Bodo Möller, and David Pointcheval. SOKE ciphersuites for password-based authentication in TLS. Work in Progress, to be published as Internet Draft, 2006.
2. Michel Abdalla, Olivier Chevassut, and David Pointcheval. One-time verifier-based encrypted key exchange. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 47–64. Springer-Verlag, January 2005.
3. Michel Abdalla, Pierre-Alain Fouque, and David Pointcheval. Password-based authenticated key exchange in the three-party setting. In Serge Vaudenay, editor, *PKC 2005*, volume 3386 of *LNCS*, pages 65–84. Springer-Verlag, January 2005.
4. Michel Abdalla and David Pointcheval. Simple password-based encrypted key exchange protocols. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 191–208. Springer-Verlag, February 2005.
5. Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155. Springer-Verlag, May 2000.
6. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS 93*, pages 62–73. ACM Press, November 1993.
7. Mihir Bellare and Phillip Rogaway. The AuthA protocol for password-based authenticated key exchange. Contributions to IEEE P1363, March 2000.
8. Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *1992 IEEE Symposium on Security and Privacy*, pages 72–84. IEEE Computer Society Press, May 1992.
9. Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In *ACM CCS 93*, pages 244–250. ACM Press, November 1993.
10. Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 156–171. Springer-Verlag, May 2000.
11. Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Security proofs for an efficient password-based key exchange. In *ACM CCS 03*, pages 241–250. ACM Press, October 2003.
12. Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. New security results on encrypted key exchange. In Feng Bao, Robert Deng, and Jianying Zhou, editors, *PKC 2004*, volume 2947 of *LNCS*, pages 145–158. Springer-Verlag, March 2004.
13. Tim Dierks and Christopher Allen. *RFC 2246 - The TLS Protocol Version 1.0*. Internet Activities Board, January 1999.
14. Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1978.
15. Liang Fang, Samuel Meder, Olivier Chevassut, and Frank Siebenlist. Secure password-based authenticated key exchange for web services. In Ernesto Damiani and Hiroshi Maruyama, editors, *Proceedings of the ACM Workshop on Secure Web Services (SWS)*, Fairfax, VA, USA, October 29, 2004.
16. Ian T. Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.
17. Ian T. Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A security architecture for computational grids. In *ACM CCS 98*, pages 83–92. ACM Press, November 1998.

18. Jonathan Katz, Rafail Ostrovsky, and Moti Yung. Forward secrecy in password-only key exchange protocols. In Stelvio Cimato, Clemente Galdi, and Giuseppe Persiano, editors, *SCN 02*, volume 2576 of *LNCS*, pages 29–44. Springer-Verlag, September 2002.
19. Stefan Lucks. Open key exchange: How to defeat dictionary attacks without encrypting public keys. In *Workshop on Security Protocols*, École Normale Supérieure, 1997.
20. Philip D. MacKenzie. The PAK suite: Protocols for password-authenticated key exchange. Technical Report 2002-46, DIMACS, 2002.
21. David Pointcheval and Jacques Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, 2000.
22. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signature and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
23. Michael Steiner, Peter Buhler, Thomas Eirich, and Michael Waidner. Secure password-based cipher suite for TLS. *ACM Transactions on Information and System Security*, 4(2):134–157, 2001.
24. David Taylor, Tom Wu, Nikos Mavroyanopoulos, and Trevor Perrin. Using SRP for TLS authentication. IETF Internet Draft, TLS Working Group, August 19, 2004.

A Equivalence of the SPCCDH and CDH Problems

Here we show that the Set Password-based Chosen-Basis Computational Diffie-Hellman $\mathsf{SPCCDH}_{g,\mathbb{G}}$ problem is equivalent to the (basic) computational Diffie-Hellman problem $\mathsf{CDH}_{g,\mathbb{G}}$. With same notations as in Section 3.1, if one has:

$$\mathsf{Succ}^{\mathsf{spccdh}}(t,s,n) \geq \frac{1}{n} + \varepsilon,$$

then, with $\nu = \max\left\{\frac{1}{n}, \frac{\varepsilon}{2}\right\}$, one has:

$$\mathsf{Succ}^{\mathsf{cdh}}(2t+\tau_e) \geq \frac{\varepsilon\nu}{2s^2} \times \left(\frac{1}{n} + \nu\right). \tag{11}$$

Proof. For proving this relation, one simply applies the splitting lemma [21]:

Lemma 2 (Splitting Lemma). *Let $S \subset A \times B$ such that $\Pr[(a,b) \in S] \geq \alpha$. For any $\beta < \alpha$, define*

$$T = \left\{(a,b) \in A \times B \;\middle|\; \Pr_{b' \in B}[(a,b') \in S] \geq \alpha - \beta\right\}.$$

Then

$$\begin{aligned} &(i) && \Pr[T] \geq \beta, \\ &(ii) && \forall (a,b) \in T, \Pr_{b' \in B}[(a,b') \in S] \geq \alpha - \beta. \end{aligned}$$

Let $\mathcal{A}$ be an adversary against the $\mathsf{SPCCDH}_{g,\mathbb{G}}$-problem, with success probability $\alpha = 1/n + \varepsilon$. Then, we can use the splitting lemma, with $\beta = \varepsilon/2$, on

$$A = \{(\omega, U, X)\} \text{ and } B = \{1, \ldots, n\} \approx \mathcal{D}.$$

Our adversary $\mathcal{B}$ receives as input a random $\mathsf{CDH}_{g,\mathbb{G}}$ instance (U, X). It chooses a random tape ω for $\mathcal{A}$: with probability greater than $\varepsilon/2$, the success probability is greater than $1/n + \varepsilon/2$, over the probability space $B = \{1, \ldots, n\}$. It is thus a multiple of $1/n$, not smaller than $1/n + \nu$, where ν is the maximum in $\{1/n, \varepsilon/2\}$. One first simply runs $\mathcal{A}$ with a random k, and with probability greater than $1/n + \nu$, one gets a first set $\mathcal{S}_1$ with $K = \mathsf{CDH}_{g,\mathbb{G}}(X, Y/U^k)$. One runs $\mathcal{A}$ again, with another random $k' \neq k$, and with probability greater than ν, one gets a second set $\mathcal{S}_2$ with $K' = \mathsf{CDH}_{g,\mathbb{G}}(X, Y/U^{k'})$. Then, $\mathsf{CDH}_{g,\mathbb{G}}(X, U) = (K/K')^{1/(k-k')}$. By choosing two elements at random in $\mathcal{S}_1$ and $\mathcal{S}_2$, one gets $\mathsf{CDH}_{g,\mathbb{G}}(X, U)$ with probability $1/s^2$.