This is the full version of the extended abstract which appears in
Proceedings of the 10th ACM Conference on Computer and Communications Security, pages 241–250
(October 27 – 30, 2003, Washington, DC, USA.)

Security Proofs for an Efficient Password-Based Key Exchange

E. Bresson[1], O. Chevassut[2], and D. Pointcheval[3]

[1] Département Cryptologie, CELAR, 35174 Bruz Cedex, France
`Emmanuel.Bresson@m4x.org`.
[2] Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
`OChevassut@lbl.gov`.
[3] CNRS–École normale supérieure, 75230 Paris Cedex 05, France
`David.Pointcheval@ens.fr`.

Abstract. Password-based key exchange schemes are designed to provide entities communicating over a public network, and sharing a (short) password only, with a session key (e.g, the key is used for data integrity and/or confidentiality). The focus of the present paper is on the analysis of very efficient schemes that have been proposed to the IEEE P1363 Standard working group on password-based authenticated key-exchange methods, but for which actual security was an open problem. We analyze the AuthA key exchange scheme and give a complete proof of its security. Our analysis shows that the AuthA protocol and its multiple modes of operation are provably secure under the computational Diffie-Hellman intractability assumption, in both the random-oracle and the ideal-cipher models.

1 Introduction

Problem. The need for secure authentication seems obvious when two entities—a client and a server—communicate on the wired-Internet, but proving an identity over a public link is complex. The method deployed by the engineers of the Secure Shell protocol (SSH) [2] to determine a client's identity to log him/her into another computer, execute commands on a remote machine, and move files from one machine to another is to ask him to type-in a password. The remote machine maintains the association between the client name and the password. Another method is to take advantage of a public-key infrastructure (PKI) to check that an entity knows the secret-key corresponding to the public-key embedded in a certificate. This method was adopted by the IETF TLS Working Group to secure the traffic between a web browser and a bank server over the wired-Internet, but work is currently under way to enrich this "transport layer" security protocol (TLS) with password-based authentication methods [18].

The primary *raison d'être* for password-based authentication is to enable clients to identify themselves to servers through a lightweight process since no security infrastructure or special hardware to carry the passwords is required. One example is when a password is used as a means to establish a secure communication channel from the computing device a human relies on to the remote machine he wants to talk to. This process, or password-authenticated key-exchange as it is often termed [6, 7, 21], provides the two computing devices with a session key to implement an authenticated communication channel within which messages sent over the wire are cryptographically protected. Humans directly benefit from this approach since they only need to remember a low-quality string (i.e. 4 decimal digits) chosen from a relatively small dictionary rather than a high-quality symmetric encryption key.

The fundamental security goal for a password-authenticated key exchange protocol to achieve is security against dictionary attacks. One can not actually prevent the adversary from guessing a value for the password and using this value in an attempt to impersonate a player. If the attack fails, the adversary can eliminate this value from the list of possible passwords. However, one would like this attack to be the only one the adversary can mount: after n active interactions with some participants the adversary should not be able to eliminate a greater number of passwords than n. Namely, a passive eavesdropping should be of no help to the adversary since an off-line exhaustive search on the password should not get any bias on the actual password. The off-line exhaustive search is called a *dictionary attack*.

The need for lightweight authentication processes is even greater in the case of the wireless-Internet. Wireless nodes are devices with particular mobility, computation and bandwidth requirements (diskless base station, cellular phone, pocket PC, palm pilot, laptop computer, base station gateway) that place severe restrictions when designing cryptographic mechanisms. The TLS protocol has been enriched with elliptic-curve cipher suites to run on low-power devices [8] and has within the WAP Forum evolved into a "transport layer" security protocol to secure mobile-commerce (WTLS) [22]. The Wired Equivalent Privacy (WEP) protocol, which is part of the IEEE 802.11 standard, does rely on high-quality symmetric encryption keys for protecting the wireless local-area network (WLAN) traffic between a mobile device equipped with a wireless ethernet-card and a fixed access point, but the WEP does not specify how the keys are established [9]. Currently, the IEEE 802.11 standard does not specify any method for key exchange.

Contributions. This paper examines the security of the AuthA password-authenticated key exchange protocol proposed to the IEEE P1363 Study Group on standard specifications for public-key cryptography [20]. Although AuthA has been conjectured cryptographically secure by its authors, it has still not been proven to resist dictionary attacks [4]. In this paper we provide a complete proof of security for the AuthA protocol. We work out our proofs by first defining the execution of AuthA in the communication model of Bellare *et al.* [3] and then adapting the proof techniques recently published by Bresson *et al.* [12] for the password-based group key exchange.

We have defined the execution of AuthA in Bellare *et al.*'s model wherein the protocol entities are modeled through oracles, and the various types of attacks are modeled by queries to these oracles. This model enables a treatment of dictionary attacks by allowing the adversary to obtain honest executions of the AuthA protocol. The security of AuthA against dictionary attacks depends on how many interactions the adversary carries out against the protocol entities rather than on the adversary's computational power. Our analysis shows that some of the AuthA modes of operation achieve provable security against dictionary attacks in both the random oracle and ideal-cipher models [3, 5] under the computational Diffie-Hellman intractability assumption.

Related Work. The IEEE P1363 Standard working group on password-based authenticated key-exchange methods [21] has been focusing on key exchange protocols wherein clients use short passwords in place of certificates to identify themselves to servers. This standardization effort has its roots in the works of Bellare *et al.* [3] and Boyko *et al.* [11], wherein formal models and security goals for password-based key agreement were first formulated. Bellare *et al.* analyzed the EKE protocol [6] (where EKE stands for *Encrypted Key Exchange*), a classical Diffie-Hellman key exchange wherein the two flows are encrypted using the password as common symmetric key. While they announced a security result of this "elegant" and efficient structure in both the random oracle and ideal-cipher models, the full proof never appeared anywhere. On the other hand, Boyko *et al.* [11] provided such a proof, but it was in another security model, the multi-party simulatability model. We thus provide a complete proof in the Bellare *et al.* security model, in a model where both a random oracle and an ideal-cipher are available.

One should note that Boyko *et al.*'s security result [11] holds in the random oracle model, while Bellare *et al.*'s one [3] holds in both the random oracle model and the ideal-cipher one together. More recent works provided password-based schemes for which security holds in the standard model only [15–17]. These are either based on general computational assumptions, or on the Decisional Diffie-Hellman problem (using a variant of the Cramer-Shoup encryption scheme [14].) While relying on a strong computational assumption, they are neither practical nor very efficient.

These provably secure schemes in the standard model are from a theoretical point of view very interesting, but fails to be practical. Ideal models (i.e. random-oracle, ideal-cipher) have thus been defined to provide alternative security results. While not being formal proofs, they give strong evidence that the schemes are not flawed. They often rely on weaker computational assumptions (e.g. the computational Diffie-Hellman problem instead of the decisional one.)

More interestingly, EKE later evolved into the proposal AuthA [4], which is formally modeled by the One-Encryption Key-Exchange (OEKE) in the present paper: only one flow is encrypted (using either a symmetric-encryption primitive or a multiplicative function as the product of a Diffie-Hellman value with a hash of the password). The advantage of such a scheme over the classical EKE, wherein the two Diffie-Hellman values are encrypted, is its easyness of integration. An OEKE cipher enables us to avoid many compatibility problems when adding password-based capabilities to existing network security protocols since the initial messages of the security protocols do not need to be modified. This argument in favor of OEKE was put forward when discussions were under way to enrich the Transport Layer Security (TLS) protocol with password-based key-exchange cipher suites [18, 19]. In a TLS One-Encryption Key-Exchange initiated by the server, the server does not need to know the client's name (a name is mapped to a password by the server using a local database) to compute and send out the server's TLS key-exchange message, but does need it to process the incoming client's TLS key-exchange message. Therefore, engineers embodied the client's name in the client's TLS key-exchange message rather than embodying it in the client's TLS hello message [18]. OEKE is thus of great practical interest, but none of the previous security analyses ever dealt with it.

Our paper is organized as follows. In Section 2, we recall the model and the definitions that should be satisfied by a password-based key exchange protocol. In Section 3, we show that OEKE, a "simplified" variant of a AuthA mode of operation, is secure. In Section 4, we build on this result to show that some of the AuthA modes of operation proposed to the IEEE P1363 Study Group are secure.

2 Model

In this section we recall the formal model for security against dictionary attacks where the adversary's capabilities are modeled through queries. In this model, the players do not deviate from the protocol and the adversary is not a player, but does control all the network communications.

2.1 Security Model

Players. We denote a *server* S and a user, or *client*, U that can participate in the key exchange protocol P. Each of them may have several *instances* called oracles involved in distinct, possibly concurrent, executions of P. We denote client instances and server instances by U^i and S^j (or by I when we consider any kind of instance).

The client and the server share a low-entropy secret pw which is (uniformly) drawn from a small dictionary Password of size N. The assumption of the uniform distribution for the password is just to make notations simpler, but everything would work with any other distribution, replacing the probability q/N by the sum of the probabilities of the q most probable passwords.

Abstract Interface. The protocol *Auth*A consists of the following algorithm:

– The *key exchange* algorithm $\textsc{KeyExch}(U^i, S^j)$ is an interactive protocol between U^i and S^j that provides the instances of U and S with a session key sk.

Queries. The adversary $\mathcal{A}$ interacts with the participants by making various queries. Let us explain the capability that each query captures:

– Execute(U^i, S^j): This query models passive attacks, where the adversary gets access to honest executions of P between U^i and S^j by eavesdropping.
– Reveal(I): This query models the misuse of the session key by instance I. The query is only available to $\mathcal{A}$ if the targetted instance actually "holds" a session key and it releases sk to $\mathcal{A}$.
– Send(I, m): This query models $\mathcal{A}$ sending a message to instance I. The adversary $\mathcal{A}$ gets back the response I generates in processing the message m according to the protocol P. A query Send$(U^i, \texttt{Start})$ initializes the key exchange algorithm, and thus the adversary receives the flow the client should send out to the server.

The Execute-query may at first seem useless since using the Send-query the adversary has the ability to carry out honest executions of P among parties. Yet the Execute-query is essential for properly dealing with dictionary attacks. The number q_s of Send-queries directly asked by the adversary does not take into account the number of Execute-queries. Therefore, q_s represents the number of flows the adversary may have built by itself, and thus the number of passwords it would have tried.

2.2 Security Notions

Freshness. The freshness notion captures the intuitive fact that a session key is not "obviously" known to the adversary. An instance is said to be **Fresh** in the current protocol execution if the instance has accepted and neither it nor the other instance with the same session tag have been asked for a Reveal-query.

The Test***-query.*** The semantic security of the session key is modeled by an additional query $\mathsf{Test}(I)$. The Test-query can be asked at most once by the adversary $\mathcal{A}$ and is only available to $\mathcal{A}$ if the attacked instance I is **Fresh**. This query is answered as follows: one flips a (private) coin b and forwards sk (the value $\mathsf{Reveal}(I)$ would output) if $b = 1$, or a random value if $b = 0$.

AKE Security. The security notions take place in the context of executing P in the presence of the adversary $\mathcal{A}$. The game $\mathbf{Game}^{\mathsf{ake}}(\mathcal{A}, P)$ is initialized by drawing a password pw from $\mathsf{Password}$, providing coin tosses to $\mathcal{A}$, all oracles, and then running the adversary by letting it asking a polynomial number of queries as described above. At the end of the game, $\mathcal{A}$ outputs its guess b' for the bit b involved in the Test-query.

We denote the **AKE advantage** as the probability that $\mathcal{A}$ correctly guesses the value of b; more precisely we define $\mathsf{Adv}_P^{\mathsf{ake}}(\mathcal{A}) = 2\Pr[b = b'] - 1$, where the probability space is over all the random coins of the adversary and all the oracles. The protocol P is said to be **AKE-secure** if $\mathcal{A}$'s advantage is negligible in the security parameter.

Authentication. Another goal of the adversary is to impersonate the client or the server. In the present paper, we focus on unilateral authentication of the client, thus we denote by $\mathsf{Succ}_P^{\mathsf{c-auth}}(\mathcal{A})$ the probability that $\mathcal{A}$ successfully impersonates a client instance in an execution of P: this means that a server would accept a key while the latter is shared with no client. The protocol P is said to be **C-Auth-secure** if such a probability is negligible in the security parameter.

2.3 Computational Diffie-Hellman Assumption

Let $\mathbb{G} = \langle g \rangle$ be a finite cyclic group of order a ℓ-bit prime number q, where the operation is denoted multiplicatively. A (t, ε)-CDH attacker in $\mathbb{G}$ is a probabilistic machine Δ running in time t such that

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

where the probability is taken over the random values x and y. The CDH-Problem is (t, ε)**-intractable** if there is no (t, ε)-attacker in $\mathbb{G}$. The CDH-assumption states that is the case for all polynomial t and any non-negligible ε.

3 One-Encryption Key Exchange

In this section, we describe OEKE, a "simplified" variant of a AuthA mode of operation [4], and prove its security in the random oracle and the ideal-cipher models. At the core of this variant resides only one flow of the basic Diffie-Hellman key exchange encrypted under the password and two protocol entities holding the same password. It therefore slightly differs from the original EKE [3, 6] in the sense that only one flow is encrypted using the password; instead of the two as usually done. But then, it is clear that at least one authentication flow has to be sent. We prove this is enough to satisfy the above security notions.

3.1 Description of the Scheme

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. Hash functions from $\{0,1\}^\star$ to $\{0,1\}^{\ell_0}$ and $\{0,1\}^{\ell_1}$ are denoted $\mathcal{H}_0$ and $\mathcal{H}_1$. A block cipher is denoted $(\mathcal{E}_k, \mathcal{D}_k)$ where $k \in \mathsf{Password}$. We also define $\bar{\mathbb{G}}$ to be equal to $\mathbb{G} \backslash \{1\}$, thus $\bar{\mathbb{G}} = \{g^x \,|\, x \in \mathbb{Z}_q^\star\}$.

$$
\begin{array}{rcl}
\underline{\textit{Client } U} & & \underline{\textit{Server } S} \\
pw & & pw \\
\mathsf{accept} \leftarrow \mathsf{false} & & \mathsf{accept} \leftarrow \mathsf{false} \\
\mathsf{terminate} \leftarrow \mathsf{false} & & \mathsf{terminate} \leftarrow \mathsf{false} \\
x \overset{R}{\leftarrow} [1, q-1] & & y \overset{R}{\leftarrow} [1, q-1] \\
X \leftarrow g^x & \xrightarrow{\quad U, X \quad} & Y \leftarrow g^y \\
Y \leftarrow \mathcal{D}_{pw}(Y^\star) & \xleftarrow{\quad S, Y^\star \quad} & Y^\star \leftarrow \mathcal{E}_{pw}(Y) \\
K_U \leftarrow Y^x & & K_S \leftarrow X^y \\
Auth \leftarrow \mathcal{H}_1(U\|S\|X\|Y\|K_U) & & \\
sk_U \leftarrow \mathcal{H}_0(U\|S\|X\|Y\|K_U) & & \\
\mathsf{accept} \leftarrow \mathsf{true} & \xrightarrow{\quad Auth \quad} & Auth \overset{?}{=} \mathcal{H}_1(U\|S\|X\|Y\|K_S) \\
 & & \text{if true, } \mathsf{accept} \leftarrow \mathsf{true} \\
 & & sk_S \leftarrow \mathcal{H}_0(U\|S\|X\|Y\|K_S) \\
\mathsf{terminate} \leftarrow \mathsf{true} & & \mathsf{terminate} \leftarrow \mathsf{true}
\end{array}
$$

Fig. 1. An execution of the protocol OEKE, run by the client U and the server S. The session key is $sk = \mathcal{H}_0(U\|S\|X\|Y\|Y^x) = \mathcal{H}_0(U\|S\|X\|Y\|X^y)$.

As illustrated on Figure 1 (with an honest execution of the OEKE protocol), the protocol runs between a client U and a server S, and the session-key space $\mathbf{SK}$ associated to this protocol is $\{0,1\}^{\ell_0}$ equipped with a uniform distribution. Client and server initially share a low-quality string pw, the password, uniformly drawn from the dictionary Password.

The protocol consists of three flows. The client chooses a random exponent x and computes the value g^x which he sends to the server. The server in turn chooses a random exponent y, computes the value g^y, and encrypts the latter under the password pw before to send it out on the wire. Upon receiving the client's flow, the server computes the Diffie-Hellman secret value g^{xy}, and from it the session key sk. Upon receiving the server's flow, the client decrypts the ciphertext, computes the Diffie-Hellman secret value, and an authentication tag $Auth$ for client-to-server unilateral authentication. The client then sends out this authenticator. If the authenticator verifies on the server side, the client and the server have successfully exchanged the session key sk.

3.2 Semantic Security

In this section, we assert that under reasonable and well-defined intractability assumptions the protocol securely distributes session keys. More precisely, in this section, we deal with the semantic security goal. We consider the unilateral authentication goal in the next section. In the proof below, we do not consider forward-secrecy, for simplicity, but the semantic security still holds in this context, with slightly different bounds. The details can be found in the Appendix D. However, remember that any security result considers concurrent executions.

Theorem 1. *Let us consider the OEKE protocol, where* $\mathbf{SK}$ *is the session-key space and* Password *is a finite dictionary of size N equipped with the uniform distribution. Let $\mathcal{A}$ be an adversary against the AKE security of OEKE within a time bound t, with less than q_s interactions with the parties and q_p passive eavesdroppings, and, asking q_h hash-queries and q_e encryption/decryption queries. Then we have*

$$\mathsf{Adv}^{\mathsf{ake}}_{\mathsf{oeke}}(\mathcal{A}) \leq 3 \times \frac{q_s}{N} + 8q_h \times \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t') + \frac{(2q_e + 3q_s + 3q_p)^2}{q-1} + \frac{q_h^2 + 4q_s}{2^{\ell_1}}.$$

where $t' \leq t + (q_s + q_p + q_e + 1) \cdot \tau_{\mathbb{G}}$, with $\tau_{\mathbb{G}}$ denoting the computational time for an exponentiation in $\mathbb{G}$. (Recall that q is the order of $\mathbb{G}$.)

This theorem shows that the OEKE protocol is secure against dictionary attacks since the advantage of the adversary essentially grows with the ratio of interactions (number of Send-queries) to the number of passwords. This is particularly significant in practice since a password may expire once a number of failed interactions has been achieved, whereas adversary's capability to enumerate passwords off-line is only limited by its computational power. Of course, this security result only holds provided that the adversary does not solve the computational Diffie-Hellman problem.

Proof (of Theorem 1). In this section we incrementally define a sequence of games starting at the real game $\mathbf{G}_0$ and ending up at $\mathbf{G}_8$.

Game $\mathbf{G}_0$***:*** This is the real attack game, in the random oracle and ideal-cipher models. Several oracles are thus available to the adversary: two hash oracles ($\mathcal{H}_0$ and $\mathcal{H}_1$), the encryption/decryption oracles ($\mathcal{E}$ and $\mathcal{D}$), and all the instances U^i and S^j (in order to cover concurrent executions). We define several events in any game $\mathbf{G}_n$:

– event S_n occurs if $b = b'$, where b is the bit involved in the Test-query, and b' is the output of the AKE-adversary;
– event $\mathsf{Encrypt}_n$ occurs if $\mathcal{A}$ submits a data it has encrypted by itself using the password;
– event Auth_n occurs if $\mathcal{A}$ submits an authenticator $Auth$ that will be accepted by the server and that has been built by the adversary itself.

By definition,

$$\mathsf{Adv}^{\mathsf{ake}}_{\mathsf{oeke}}(\mathcal{A}) = 2\Pr[\mathsf{S}_0] - 1. \tag{1}$$

In the games below, we furthermore assume that when the game aborts or stops with no answer b' outputted by the adversary $\mathcal{A}$, we choose this bit b' at random, which in turn defines the actual value of the event S_k. Moreover, if the adversary has not finished playing the game after q_s Send-queries or lasts for more than time t, we stop the game (and choose a random bit b'), where q_s and t are predetermined upper-bounds.

Game $\mathbf{G}_1$***:*** In this game, we simulate the hash oracles ($\mathcal{H}_0$ and $\mathcal{H}_1$, but also two additional hash functions $\mathcal{H}_2 : \{0,1\}^\star \to \{0,1\}^{\ell_2}$ and $\mathcal{H}_3 : \{0,1\}^\star \to \{0,1\}^{\ell_3}$, with $\ell_2 = \ell_0$ and $\ell_3 = \ell_1$, that will appear in Game $\mathbf{G}_7$) and the encryption/decryption oracles, as usual by maintaining a hash list $\Lambda_{\mathcal{H}}$ (and another list $\Lambda_{\mathcal{A}}$ containing the hash-queries asked by the adversary itself) and an encryption list $\Lambda_{\mathcal{E}}$ (see Figure 2) We also simulate all the instances, as the real players would do, for the Send-queries (see Figure 3) and for the Execute, Reveal and Test-queries (see Figure 4).

From this simulation, we easily see that the game is perfectly indistinguishable from the real attack, unless the permutation property of $\mathcal{E}$ or $\mathcal{D}$ does not hold. One could have avoided collisions but this happens with probability at most $q_{\mathcal{E}}^2/2(q-1)$ since $|\bar{\mathbb{G}}| = (q-1)$, where $q_{\mathcal{E}}$ is the size of $\Lambda_{\mathcal{E}}$:

$$|\Pr[\mathsf{S}_1] - \Pr[\mathsf{S}_0]| \le \frac{q_{\mathcal{E}}^2}{2(q-1)}. \tag{2}$$

Game $\mathbf{G}_2$***:*** We define game $\mathbf{G}_2$ by modifying the way the server processes the Send-queries so that the adversary will be the only one to encrypt data. We use the following rule:

▶**Rule S1**$^{(2)}$ – Choose a random $Y^\star \in \bar{\mathbb{G}}$ and compute $Y = \mathcal{D}_{pw}(Y^\star)$. Look for the record $(pw, Y, \varphi, *, Y^\star)$ in the list $\Lambda_{\mathcal{E}}$ to define φ (we thus have $Y = g^\varphi$), and finally compute $K_S = X^\varphi$.

The two games $\mathbf{G}_2$ and $\mathbf{G}_1$ are perfectly indistinguishable unless $\varphi = \perp$. This happens when $Y^\star$ has been previously obtained as the ciphertext returned by an encryption-query. Note that this may happen when processing a Send-query, but also during a passive simulation when processing an Execute-query:

$$|\Pr[\mathsf{S}_2] - \Pr[\mathsf{S}_1]| \le \frac{q_S q_{\mathcal{E}}}{q-1}, \tag{3}$$

For a hash-query $\mathcal{H}_i(q)$ (with $i \in \{0,1,2,3\}$), such that a record (i,q,r) appears in $\Lambda_\mathcal{H}$, the answer is r. Otherwise the answer r is defined according to the following rule:

▶**Rule** $\mathcal{H}^{(1)}$ – Choose a random element $r \in \{0,1\}^{\ell_i}$.

The record (i,q,r) is added to $\Lambda_\mathcal{H}$. If the query is directly asked by the adversary, one adds (i,q,r) to $\Lambda_\mathcal{A}$.

For an encryption-query $\mathcal{E}_k(Z)$, such that a record $(k,Z,*,*,Z^\star)$ appears in $\Lambda_\mathcal{E}$, the answer is $Z^\star$. Otherwise the answer $Z^\star$ is defined according to the following rule:

▶**Rule** $\mathcal{E}^{(1)}$ – Choose a random element $Z^\star \in \bar{\mathbb{G}}$.

Then one adds the record $(k,Z,\perp,\mathcal{E},Z^\star)$ to $\Lambda_\mathcal{E}$.

For a decryption-query $\mathcal{D}_k(Z^\star)$, such that a record $(k,Z,*,*,Z^\star)$ appears in $\Lambda_\mathcal{E}$, the answer is Z. Otherwise, one applies the following rule to obtain the answer Z:

▶**Rule** $\mathcal{D}^{(1)}$ – Choose a random element $\varphi \in \mathbb{Z}_q^\star$, compute the answer $Z = g^\varphi$ and add the record $(k,Z,\varphi,\mathcal{D},Z^\star)$ to $\Lambda_\mathcal{E}$.

Fig. 2. Simulation of the random oracles, and the encryption/decryption oracles

We answer to the Send-queries to the client as follows:

- A $\mathsf{Send}(U^i, \mathtt{Start})$-query is processed according to the following rule:
  ▶**Rule U1**$^{(1)}$ – Choose a random exponent $\theta \in \mathbb{Z}_q^\star$ and compute $X = g^\theta$.
  Then the query is answered with U, X, and the client instance goes to an expecting state.
- If the client instance U^i is in an expecting state, a query $\mathsf{Send}(U^i, (S, Y^\star))$ is processed by computing the session key and producing an authenticator. We apply the following rules:
  ▶**Rule U2**$^{(1)}$ – Compute $Y = \mathcal{D}_{pw}(Y^\star)$ and $K_U = Y^\theta$.
  ▶**Rule U3**$^{(1)}$ – Compute the authenticator $Auth = \mathcal{H}_1(U\|S\|X\|Y\|K_U)$ and the session key $sk_U = \mathcal{H}_0(U\|S\|X\|Y\|K_U)$.
  Finally the query is answered with $Auth$, the client instance accepts and terminates. Our simulation also adds $((U,X),(S,Y^\star),Auth)$ to Λ_Ψ. The variable Λ_Ψ keeps track of the exchanged messages.

We answer to the Send-queries to the server as follows:

- A $\mathsf{Send}(S^j, (U, X))$-query is processed according to the following rule:
  ▶**Rule S1**$^{(1)}$ – Choose a random exponent $\varphi \in \mathbb{Z}_q^\star$, compute $Y = g^\varphi$, $Y^\star = \mathcal{E}_{pw}(Y)$ and $K_S = X^\varphi$.
  Finally, the query is answered with $S, Y^\star$ and the server instance goes to an expecting state.
- If the server instance S^j is in an expecting state, a query $\mathsf{Send}(S^j, H)$ is processed according to the following rules:
  ▶**Rule S2**$^{(1)}$ – Compute $H' = \mathcal{H}_1(U\|S\|X\|Y\|K_S)$, and check whether $H = H'$. If the equality does not hold, the server instance terminates without accepting.
  If equality holds, the server instance accepts and goes on, applying the following rule:
  ▶**Rule S3**$^{(1)}$ – Compute the session key $sk_S = \mathcal{H}_0(U\|S\|X\|Y\|K_S)$.
  Finally, the server instance terminates.

Fig. 3. Simulation of the Send-queries

An $\mathsf{Execute}(U^i, S^j)$-query is processed using successively the simulations of the Send-queries: $(U,X) \leftarrow \mathsf{Send}(U^i, \mathtt{Start})$, $(S, Y^\star) \leftarrow \mathsf{Send}(S^j, (U,X))$ and $Auth \leftarrow \mathsf{Send}(U^i, (S, Y^\star))$, and outputting the transcript $((U,X),(S,Y^\star),Auth)$.

A $\mathsf{Reveal}(I)$-query returns the session key (sk_U or sk_S) computed by the instance I (if the latter has accepted).

A $\mathsf{Test}(I)$-query first gets sk from $\mathsf{Reveal}(I)$, and flips a coin b. If $b = 1$, we return the value of the session key sk, otherwise we return a random value drawn from $\{0,1\}^{\ell_0}$.

Fig. 4. Simulation of the Execute, Reveal and Test-queries

where q_S is the number of involved server instances: $q_S \leq q_s + q_p$. Furthermore note that from now, only the adversary may ask encryption queries, since the server is simulated using the decryption oracle.

Game $\mathbf{G}_3$***:*** In this game, we avoid collisions amongst the hash queries asked by the adversary to $\mathcal{H}_1$, amongst the passwords and the ciphertexts, and amongst the output of the Send-queries. We play the game in a way that: no collision has been found by the adversary for $\mathcal{H}_1$; no encrypted data corresponds to multiple identical plaintext; at most one password corresponds to each plaintext-ciphertext pair; abort if two instances of the server have used the same random values. This will help us later on to prove Lemma 2, the key step in proving Theorem 1. We use the following rules:

►**Rule** $\mathcal{H}^{(3)}$ – Choose a random element $r \in \{0,1\}^{\ell_i}$. If $i = 1$, this query is directly asked by the adversary, and $(1, *, r) \in \Lambda_{\mathcal{A}}$, then we abort the game.

Then, for any H, $\#\{(1, *, H) \in \Lambda_{\mathcal{A}}\} \leq 1$. But this rule may make the game to abort with probability bounded by $q_h^2/2^{\ell_1+1}$

►**Rule** $\mathcal{E}^{(3)}$ – Choose a random element $Z^\star \in \bar{\mathbb{G}}$. If $(*, *, \perp, \mathcal{E}, Z^\star) \in \Lambda_{\mathcal{E}}$, we abort the game.

Then, for any $Z^\star$, $\#\{(*, *, \perp, \mathcal{E}, Z^\star) \in \Lambda_{\mathcal{E}}\} \leq 1$. But this rule may make the game to abort with probability bounded by $q_{\mathcal{E}}^2/2(q-1)$.

►**Rule** $\mathcal{D}^{(3)}$ – Choose a random element $\varphi \in \mathbb{Z}_q^\star$ and compute $Z = g^\varphi$. If $(*, Z, *, *, Z^\star) \in \Lambda_{\mathcal{E}}$, we abort the game. Otherwise, we add the record $(k, Z, \varphi, \mathcal{D}, Z^\star)$ to $\Lambda_{\mathcal{E}}$.

Then, for any pair $(Z, Z^\star)$, $\#\{(*, Z, *, *, Z^\star) \in \Lambda_{\mathcal{E}}\} \leq 1$. But this rule may make the game to abort with probability bounded by $q_{\mathcal{E}}^2/2(q-1)$.

►**Rule S1**$^{(3)}$ – Choose a random $Y^\star \in \bar{\mathbb{G}}$. If $(*, Y^\star) \in \Lambda_S$, abort the game, otherwise add the record $(j, Y^\star)$ to Λ_S. Then, compute $Y = \mathcal{D}_{pw}(Y^\star)$, look for the record $(pw, Y, \varphi, *, Y^\star)$ in $\Lambda_{\mathcal{E}}$ to define φ (we thus have $Y = g^\varphi$), and compute $K_S = X^\varphi$. The variable Λ_S keeps track of the messages sent out by the server S.

Then, there is no collision among the $Y^\star$ outputted by the server instances (and thus the used Y). But this rule may make the game to abort with probability bounded by the birthday paradox, $q_S^2/2(q-1)$, where q_S is again the number of involved server instances.

The two games $\mathbf{G}_3$ and $\mathbf{G}_2$ are perfectly indistinguishable unless one of the above rules make the game to abort:

$$|\Pr[\mathsf{S}_3] - \Pr[\mathsf{S}_2]| \leq \frac{2q_{\mathcal{E}}^2 + q_S^2}{2(q-1)} + \frac{q_h^2}{2^{\ell_1+1}}. \tag{4}$$

Game $\mathbf{G}_4$***:*** We define game $\mathbf{G}_4$ by aborting the executions wherein the adversary may have guessed the password and used it to send an encrypted data to the client. We achieve this aim by modifying the way the client processes the queries. We use the following rule:

►**Rule U2**$^{(4)}$ – Look for $(pw, *, \perp, \mathcal{E}, Y^\star) \in \Lambda_{\mathcal{E}}$. If the record is found, define $\mathsf{Encrypt}_4$ as true and abort the game. Otherwise, compute $Y = \mathcal{D}_{pw}(Y^\star)$ and $K_U = Y^\theta$.

The two games $\mathbf{G}_4$ and $\mathbf{G}_3$ are perfectly indistinguishable unless event $\mathsf{Encrypt}_4$ occurs:

$$|\Pr[\mathsf{S}_4] - \Pr[\mathsf{S}_3]| \leq \Pr[\mathsf{Encrypt}_4]. \tag{5}$$

Game $\mathbf{G}_5$***:*** We define game $\mathbf{G}_5$ by aborting the executions wherein the adversary may have been lucky in guessing the authenticator (that is, without asking the corresponding hash query). We reach this aim by modifying the way the server processes the queries:

►**Rule S2**$^{(5)}$ – Check whether $H = H'$, where $H' = \mathcal{H}_1(U\|S\|X\|Y\|K_S)$. If the equality does hold, check if $(1, U\|S\|X\|Y\|K_S, H) \in \Lambda_{\mathcal{A}}$ or $((U, X), (S, Y^\star), H) \in \Lambda_\Psi$. If these two latter tests fail, then reject the authenticator: terminate, without accepting. If this rule does not make the server to terminate, the server accepts and moves on.

This rule ensures that all accepted authenticators will come from either the simulator, or an adversary that has correctly decrypted $Y^\star$ into Y, (computed K_S) and asked the query to the oracle $\mathcal{H}_1$. The two games $\mathbf{G}_5$ and $\mathbf{G}_4$ are perfectly indistinguishable unless the server rejects a valid authenticator. Since Y did not appear in a previous session (since the Game $\mathbf{G}_3$), this happens only if the authenticator had been correctly guessed by the adversary without asking $\mathcal{H}_1(U\|S\|X\|Y\|K_S)$:

$$|\Pr[\mathsf{Encrypt}_5] - \Pr[\mathsf{Encrypt}_4]| \le \frac{q_s}{2^{\ell_1}} \qquad |\Pr[\mathsf{S}_5] - \Pr[\mathsf{S}_4]| \le \frac{q_s}{2^{\ell_1}}. \tag{6}$$

Game* $\mathbf{G}_6$*: We define game $\mathbf{G}_6$ by aborting the executions wherein the adversary may have guessed the password (that is the adversary has correctly decrypted $Y^\star$ into Y) and then used it to build and send a valid authenticator to the server. We reach this aim by modifying the way the server processes the queries:

- ▶**Rule S2**$^{(6)}$ – Check if $((U,X),(S,Y^\star),H) \in \Lambda_\Psi$. If this is not the case, then reject the authenticator: terminate, without accepting. Check if $(1, U\|S\|X\|Y\|*, H) \in \Lambda_\mathcal{A}$. If this is the case, we define the event Auth'_6 to be true, and abort the game.

This rule ensures that all accepted authenticators come from the simulator. The two games $\mathbf{G}_6$ and $\mathbf{G}_5$ are perfectly indistinguishable unless either $(1, U\|S\|X\|Y\|K_S, H) \in \Lambda_\mathcal{A}$ or $(1, U\|S\|X\|Y\|*, H) \in \Lambda_\mathcal{A}$, which both lead to Auth'_6 to be true:

$$|\Pr[\mathsf{Encrypt}_6] - \Pr[\mathsf{Encrypt}_5]| \le \Pr[\mathsf{Auth}'_6] \qquad |\Pr[\mathsf{S}_6] - \Pr[\mathsf{S}_5]| \le \Pr[\mathsf{Auth}'_6]. \tag{7}$$

Game* $\mathbf{G}_7$*: In this game, we do no compute the authenticator $Auth$ and the session key sk using the oracles $\mathcal{H}_0$ and $\mathcal{H}_1$, but using the private oracles $\mathcal{H}_2$ and $\mathcal{H}_3$ so that the values $Auth$ and sk are completely independent from $\mathcal{H}_0$ and $\mathcal{H}_1$, but also Y, pw and any of K_U or K_S. We reach this aim by using the following rules:

- ▶**Rule U3**$^{(7)}$ – Compute the session key $sk_U = \mathcal{H}_2(U\|S\|X\|Y^\star)$ and the authenticator $Auth = \mathcal{H}_3(U\|S\|X\|Y^\star)$.

- ▶**Rule S3**$^{(7)}$ – Compute the session key $sk_S = \mathcal{H}_2(U\|S\|X\|Y^\star)$.

Since we do no longer need to compute the values K_U and K_S, we can also simplify the way client and server process the queries:

- ▶**Rule U2**$^{(7)}$ – Look for a record $(pw, *, \perp, \mathcal{E}, Y^\star)$ in $\Lambda_\mathcal{E}$. If the record is found, we define $\mathsf{Encrypt}_7$ as true and abort the game.

- ▶**Rule S1**$^{(7)}$ – Choose a random $Y^\star \in \bar{\mathbb{G}}$. If $(*, Y^\star) \in \Lambda_S$, one aborts the game, otherwise adds the record $(j, Y^\star)$ to Λ_S. Then, compute $Y = \mathcal{D}_{pw}(Y^\star)$.

The games $\mathbf{G}_7$ and $\mathbf{G}_6$ are indistinguishable unless the following event AskH occurs: $\mathcal{A}$ queries the hash functions $\mathcal{H}_0$ or $\mathcal{H}_1$ on $U\|S\|X\|Y\|K_U$ or on $U\|S\|X\|Y\|K_S$, that is on the common value $U\|S\|X\|Y\|\mathsf{CDH}(X,Y)$:

$$\begin{gathered}|\Pr[\mathsf{Encrypt}_7] - \Pr[\mathsf{Encrypt}_6]| \le \Pr[\mathsf{AskH}_7] \qquad |\Pr[\mathsf{S}_7] - \Pr[\mathsf{S}_6]| \le \Pr[\mathsf{AskH}_7] \\ \left|\Pr[\mathsf{Auth}'_7] - \Pr[\mathsf{Auth}'_6]\right| \le \Pr[\mathsf{AskH}_7].\end{gathered} \tag{8}$$

Lemma 2. *The probabilities of the events* S_7, $\mathsf{Encrypt}_7$, *and* Auth'_7 *in game* $\mathbf{G}_7$ *can be upper-bounded by the following values:*

$$\Pr[\mathsf{S}_7] = \frac{1}{2} \qquad \Pr[\mathsf{Encrypt}_7] \le \frac{q_s}{2N} \qquad \Pr[\mathsf{Auth}'_7] \le \frac{q_s}{2N}. \tag{9}$$

Proof. The formal proof of this lemma can be found in the Appendix A.1. The main idea in simulating this game is to choose the password pw at the end of the game. The password pw is in fact only needed to determine whether the events $\mathsf{Encrypt}_7$ or Auth'_7 have occurred, and it turns out that determining whether these events have occurred can be postponed until the time limit has been reached or the adversary has asked q_s queries. The probabilities of $\mathsf{Encrypt}_7$ or Auth'_7 can then be easily upper-bounded since no information, in the information theoretical sense, about the password pw is known by the adversary along this simulation. □

Game* $\mathbf{G}_8$*: In this game, we simulate the executions using the random self-reducibility of the Diffie-Hellman problem, given one CDH instance (A, B). We do not need to known the values of θ and φ, since the values K_U or K_S are no longer needed to compute the authenticator and the session keys:

- ▶**Rule U1**$^{(8)}$ – Choose a random element $\alpha \in \mathbb{Z}_q^\star$, and compute $X = A^\alpha$. Also add the record (α, X) to Λ_A.
- ▶**Rule** $\mathcal{D}^{(8)}$ – Choose a random element $\beta \in \mathbb{Z}_q^\star$, and compute the answer $Z = B^\beta$. Also add the record (β, Z) to Λ_B. If $(*, Z, *, *, Z^\star) \in \Lambda_{\mathcal{E}}$ then we abort the game; otherwise we add the record $(k, Z, \perp, \mathcal{D}, Z^\star)$ to $\Lambda_{\mathcal{E}}$.

$$\Pr[\mathsf{AskH}_8] = \Pr[\mathsf{AskH}_7]. \tag{10}$$

Remember that AskH_8 means that the adversary $\mathcal{A}$ had queried the random oracles $\mathcal{H}_0$ or $\mathcal{H}_1$ on $U\|S\|X\|Y\|Z$, where $Z = \mathsf{CDH}(X, Y)$. By picking randomly in the $\Lambda_{\mathcal{A}}$-list we can get the Diffie-Hellman secret value with probability $1/q_h$. This is a triple $(X, Y, \mathsf{CDH}(X, Y))$. We can then simply look in the lists Λ_A and Λ_B to find the values α and β such that $X = A^\alpha$ and $Y = B^\beta$:

$$\mathsf{CDH}(X, Y) = \mathsf{CDH}(A^\alpha, B^\beta) = \mathsf{CDH}(A, B)^{\alpha\beta}.$$

Thus:

$$\Pr[\mathsf{AskH}_8] \le q_h \mathsf{Succ}_{\mathbb{G}}^{\mathsf{cdh}}(t'). \tag{11}$$

This concludes the proofs (the details of the computations can be found in the Appendix A.2. Simply note that $q_{\mathcal{E}}$ is the size of $\Lambda_{\mathcal{E}}$, which contains all the encryption/decryption queries directly asked by the adversary, but also all the decryption queries made by our simulation: at most one per Send-query (direct or through $\mathsf{Execute}$-queries), which makes $q_{\mathcal{E}} \le q_e + q_s + q_p$. Similarly, q_S is the number of involved server instances, and thus $q_S \le q_s + q_p$. Furthermore, one can easily see that in this last game, $t' \le t + (q_s + q_p + q_e + 1) \cdot \tau_{\mathbb{G}}$. □

3.3 Unilateral Authentication

The following theorem shows that the OEKE protocol furthermore ensures authentication from client to server, in the sense that a server instance will never accept an authenticator that has not actually been sent by the corresponding/expected client instance with probability significantly greater than q_s/N.

Theorem 3. *Let us consider the* OEKE *protocol, where* **SK** *is the session-key space and* $\mathsf{Password}$ *a finite dictionary of size N equipped with the uniform distribution. Let $\mathcal{A}$ be an adversary against the AKE security of* OEKE *within a time bound t, with less than q_s interactions with the parties and q_p passive eavesdroppings, and, asking q_h hash-queries and q_e encryption/decryption queries. Then we have*

$$\mathsf{Adv}_{\mathsf{oeke}}^{\mathsf{c-auth}}(\mathcal{A}) \le \frac{3}{2} \times \frac{q_s}{N} + 3q_h \times \mathsf{Succ}_{\mathbb{G}}^{\mathsf{cdh}}(t') + \frac{(2q_e + 3q_s + 3q_p)^2}{2(q-1)} + \frac{q_h^2 + 4q_s}{2^{\ell_1+1}}.$$

where $t' \le t + (q_s + q_p + q_e + 1)\tau_{\mathbb{G}}$, with $\tau_{\mathbb{G}}$ denoting the computational time for an exponentiation in $\mathbb{G}$. (Recall that q is the order of $\mathbb{G}$.)

Proof. The proof is similar to the previous one. But one can find more details in the Appendix B. □

$$
\begin{array}{rcl}
\textit{Client} & & \textit{Server} \\
pw_U & & pw_S = g^{pw_U} \\
\mathsf{accept} \leftarrow \mathsf{false} & & \mathsf{accept} \leftarrow \mathsf{false} \\
\mathsf{terminate} \leftarrow \mathsf{false} & & \mathsf{terminate} \leftarrow \mathsf{false} \\
x \overset{R}{\leftarrow} [1, q-1] & & y \overset{R}{\leftarrow} [1, q-1] \\
X \leftarrow g^x, \quad X^\star \leftarrow X & & Y \leftarrow g^y, \quad Y^\star \leftarrow \mathcal{E}_{pw_S}(Y) \\
& \xrightarrow{U, X^\star} \cdots \xrightarrow{U, \underline{X^\star}} & \\
& \xleftarrow{S, \underline{Y^\star}} \cdots \xleftarrow{S, Y^\star} & \\
\underline{Y} \leftarrow \mathcal{D}_{pw_S}(\underline{Y^\star}), \quad K_U \leftarrow \underline{Y}^x & & \underline{X} \leftarrow \underline{X^\star}, \quad K_S \leftarrow \underline{X}^y \\
\mathsf{PW}_U \leftarrow \underline{Y}^{pw_U} & & \mathsf{PW}_S \leftarrow {pw_S}^y \\
MK_U \leftarrow \mathcal{H}(U \| S \| X \| \underline{Y} \| K_U) & & \\
Auth \leftarrow \mathcal{H}(MK_U \| \mathsf{PW}_U) & & \\
sk_U \leftarrow \mathcal{H}(MK_U \| 0) & & \\
\mathsf{accept} \leftarrow \mathsf{true} & & \\
& \xrightarrow{Auth} \cdots \xrightarrow{\underline{Auth}} & \\
& & MK_S \leftarrow \mathcal{H}(U \| S \| \underline{X} \| Y \| K_S) \\
& & \underline{Auth} \overset{?}{=} \mathcal{H}(MK_S \| \mathsf{PW}_S) \\
& & \quad \text{if true, } \mathsf{accept} \leftarrow \mathsf{true} \\
& & sk_S \leftarrow \mathcal{H}(MK_S \| 0) \\
\mathsf{terminate} \leftarrow \mathsf{true} & & \mathsf{terminate} \leftarrow \mathsf{true}
\end{array}
$$

Fig. 5. The AuthA protocol run by the client U and the server S – The session key for U is $sk_U = \mathcal{H}(\mathcal{H}(U \| S \| X \| \underline{Y} \| \underline{Y}^x) \| 0)$. The session key for S is $sk_S = \mathcal{H}(\mathcal{H}(U \| S \| \underline{X} \| Y \| \underline{X}^y) \| 0)$.

4 Applications

We describe some applications of our security results. We first show that some of the AuthA modes of operations [4] proposed to the IEEE P1363 Standard working group encompass particular cases of OEKE. Then, we make the ideal-cipher model more concrete.

4.1 Verifier-based Key Exchange

The AuthA protocol standardized by the IEEE organization is slightly different from our protocol since client and server do not share a password pw. The AuthA has an added mechanism preventing an adversary corrupting the password table of a server from impersonating a client at once. The AuthA protocol takes advantage of the asymmetric cryptography principles when generating the passwords hold by the client and the server. The client holds a derived password $pw_U = \mathcal{H}'(U \| S \| \mathsf{PW})$ (where PW is the actual password, and pw_U has the same entropy but in $\mathbb{Z}_q^\star$) and the server holds a value pw_S derived from the latter password as follows $pw_S = g^{pw_U}$. It has the same entropy as PW too. It is then straightforward to modify our protocol to make use of these values pw_U and pw_S rather than just the shared password pw (see Figure 5): pw_S plays the role of the common password, and

$$\mathcal{H}_0(U \| S \| X \| Y \| Z) \leftarrow \mathcal{H}(\mathcal{H}(U \| S \| X \| Y \| Z) \| 0) \quad \mathcal{H}_1(U \| S \| X \| Y \| Z) \leftarrow \mathcal{H}(\mathcal{H}(U \| S \| X \| Y \| Z) \| Y^{pw_U}).$$

As a consequence, one can claim exactly the same security results about this scheme as the ones stated in the Theorems 1 and 3. More details can be found in the Appendix C.

4.2 The AuthA Modes of Operation

When engineers choose a password-based key exchange scheme, they take into account its security, computation and communication efficiency, and easiness of integration. Since they do not all face the same computing environment, they may want to operate the AuthA protocol in different ways: encrypt both flows of the basic Diffie-Hellman key exchange; achieve mutual-authentication; the server sends out the first protocol flow. These different ways have already been described in [4] and do not seem to alter the security of the AuthA protocol. But more precise security analyses similar to the above ones should be performed before actually using the other modes.

4.3 Instantiating the Encryption Function

It is clear that a simple block-cipher can not be used in place of the ideal-cipher required by the security result. We indeed need permutations onto $\mathbb{G}$ for all the secret keys, otherwise partition attacks can be mounted [10]. In specific cases where the encoding of the elements is compact, on can use the iterated technique [1]: one encrypts the element, and reencrypts the result, until one finally falls in the group $\mathbb{G}$. Decryption operates the same way. With well-chosen elliptic curves, the average number of iterations can be bounded by 2. Furthermore, the size of the blocks can thus be less than 256 bits. However, one must be careful in the implementation to prevent timing attacks.

A promising avenue is to also instantiate the encryption primitive as the product of a Diffie-Hellman value with a hash of the password, as suggested in AuthA [4]. Preliminary investigations have shown that this multiplicative function leads to a password-based key-exchange scheme secure in the random-oracle model only [13].

5 Conclusion

The reductions presented in this paper are not optimal, but our intend was to present easy to read, understand and meaningful proofs rather than very efficient ones. We think that the terms $3q_s/2N$ or $3q_s/N$ can be improved to q_s/N, but the proof would then in turn becomes very intricate. For technical reasons the hash function $\mathcal{H}_1$ used to build the authenticator has to be collision-resistant in our proofs, but the authors of AuthA [4] suggest to use a 64-bit authenticator. This may turn out to be enough in practice, but the proof presented in the paper would then need to be modified. It, however, seems a bad idea to use the same hash function $\mathcal{H}$ everywhere in AuthA.

Acknowledgments

The second 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-51868. Disclaimer available at `http://www-library.lbl.gov/disclaimer`.

References

1. M. Bellare, A. Boldyreva, A. Desai, and D. Pointcheval. Key-Privacy in Public-Key Encryption. In *Asiacrypt '01*, LNCS 2248, pages 566–582. Springer-Verlag, Berlin, 2001.
2. M. Bellare and T. Kohno and C. Namprempre. Authenticated Encryption in SSH: Provably Fixing the SSH Binary Packet Protocol. In *Proc. of the 9th CCS*. ACM Press, New York, 2002.
3. M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated Key Exchange Secure Against Dictionary Attacks. In *Eurocrypt '00*, LNCS 1807, pages 139–155. Springer-Verlag, Berlin, 2000.
4. M. Bellare and P. Rogaway. The AuthA Protocol for Password-Based Authenticated Key Exchange. Contributions to IEEE P1363. March 2000. Available from `http://grouper.ieee.org/groups/1363/`.
5. M. Bellare and P. Rogaway. Random Oracles Are Practical: a Paradigm for Designing Efficient Protocols. In *Proc. of the 1st CCS*, pages 62–73. ACM Press, New York, 1993.
6. S. M. Bellovin and M. Merritt. Encrypted Key Exchange: Password-Based Protocols Secure against Dictionary Attacks. In *Proc. of the Symposium on Security and Privacy*, pages 72–84. IEEE, 1992.
7. S. M. Bellovin and M. Merritt. Augmented Encrypted Key Exchange: A Password-Based Protocol Secure against Dictionary Attacks and Password File Compromise. In *Proc. of the 1st CCS*, pages 244–250. ACM Press, New York, 1993.

8. S. Blake-Wilson, V. Gupta, C. Hawk, and B. Moeller. ECC Cipher Suites for TLS, February 2002. IEEE RFC 20296.
9. N. Borisov, I. Goldberg, and D. Wagner. Intercepting Mobile Communications: The Insecurity of 802.11. In *Proc. of ACM International Conference on Mobile Computing and Networking (MobiCom'01)*, 2001.
10. C. Boyd, P. Montague, and K. Nguyen. Elliptic Curve Based Password Authenticated Key Exchange Protocols. In *ACISP '01*, LNCS 2119, pages 487–501. Springer-Verlag, Berlin, 2001.
11. V. Boyko, P. MacKenzie, and S. Patel. Provably Secure Password Authenticated Key Exchange Using Diffie-Hellman. In *Eurocrypt '00*, LNCS 1807, pages 156–171. Springer-Verlag, Berlin, 2000.
12. E. Bresson, O. Chevassut, and D. Pointcheval. Group Diffie-Hellman Key Exchange Secure against Dictionary Attacks. In *Asiacrypt '02*, LNCS 2501, pages 497–514. Springer-Verlag, Berlin, 2002.
13. E. Bresson, O. Chevassut, and D. Pointcheval. Encrypted Key Exchange using Mask Generation Function. Work in progress.
14. R. Cramer and V. Shoup. A Practical Public Key Cryptosystem Provably Secure against Adaptive Chosen Ciphertext Attack. In *Crypto '98*, LNCS 1462, pages 13–25. Springer-Verlag, Berlin, 1998.
15. O. Goldreich and Y. Lindell. Session-Key Generation Using Human Passwords Only. In *Crypto '01*, LNCS 2139, pages 408–432. Springer-Verlag, Berlin, 2001.
16. J. Katz, R. Ostrovsky, and M. Yung. Efficient Password-Authenticated Key Exchange Using Human-Memorizable Passwords. In *Eurocrypt '01*, LNCS 2045, pages 475–494. Springer-Verlag, Berlin, 2001.
17. J. Katz, R. Ostrovsky, and M. Yung. Forward Secrecy in Password-only Key Exchange Protocols. In *Proc. of SCN '02*, 2002.
18. M. Steiner, P. Buhler, T. Eirich, and M. Waidner. Secure Password-Based Cipher Suite for TLS. *ACM Transactions on Information and System Security (TISSEC)*, 4(2):134–157, 2001.
19. D. Taylor. Using SRP for TLS Authentication, november 2002. Internet Draft.
20. IEEE Standard 1363–2000. Standard Specifications for Public Key Cryptography. IEEE. Available from `http://grouper.ieee.org/groups/1363`, August 2000.
21. IEEE Standard 1363.2 Study Group. Password-Based Public-Key Cryptography. Available from `http://grouper.ieee.org/groups/1363/passwdPK`.
22. Wireless Application Protocol. Wireless Transport Layer Security Specification, February 2000. *WAP TLS, WAP-199 WTLS.*

A Complements for the Proof of Theorem 1

A.1 Proof of Lemma 2

Game $\mathbf{G}_7$***:*** In this game, we compute the authenticator sk_U and the session key sk_S using the private oracles $\mathcal{H}_2$ and $\mathcal{H}_3$ as depicted on Figure 6. Generating these values by querying the private oracles only X and $Y^\star$ enable us to no longer need to compute the values Y, K_U, and K_S for the simulation, but just to compute them at the end with the actual value of pw for defining the events $\mathsf{Encrypt}_7$ and Auth'_7.

The **Rule U2**$^{(7)}$, **Rule S1**$^{(7)}$ and **Rule S2**$^{(7)}$ can indeed be rewritten as rules that do not need the password along the simulation, but only make use of it at the end of the simulation. One can easily see on Figure 7 that the **Rule U2+**$^{(7)}$ and **Rule S2+**$^{(7)}$ are not useful for the simulation, but that they are only useful to determine whether events $\mathsf{Encrypt}_7$ or Auth'_7 occurred. They can thus be postponed until the adversary has asked q_s queries, or time limit expired. But then, one can note that the password pw is not used anymore, until these last rules are proceeded: one can run the simulation, without any password, and just choose it before processing these two rules.

Let us denote by $R(U)$ the set of $Y^\star$ received by a client instance, and by $R(S)$ the set of $(H, Y^\star)$ used by a server instance. From an information theoretical point of view, since we have avoided collisions in the Game $\mathbf{G}_3$,

$$\Pr[\mathsf{Encrypt}_7] = \Pr_{pw}[\exists Y^\star \in R(U), (pw, *, \perp, \mathcal{E}, Y^\star) \in \Lambda_{\mathcal{E}}] \leq \frac{\#R(U)}{N}$$

$$\Pr[\mathsf{Auth}'_7] = \Pr_{pw}[\exists (H, Y^\star) \in R(S), Y \leftarrow \mathcal{D}_{pw}(Y^\star), (1, U\|S\|X\|Y\|*, H) \in \Lambda_{\mathcal{A}}] \leq \frac{\#R(S)}{N}.$$

By definition of the sets $R(U)$ and $R(S)$, since $Y^\star$ is received in the second query to the user, and H in the second query to the server, the cardinalities are both upper-bounded by $q_s/2$.

Moreover, the session keys are random, independent from any other data (from an information theoretical point of view, since $\mathcal{H}_2$ and $\mathcal{H}_3$ are private random oracles). Then, $\Pr[\mathsf{S}_7] = 1/2$. □

We answer to the Send-queries to the client as follows:

- A Send$(U^i, \texttt{Start})$-query is processed according to the following rule:
  - ►**Rule U1**$^{(7)}$ – Choose a random exponent $\theta \in \mathbb{Z}_q^\star$ and compute $X = g^\theta$.

  Then the query is answered with U, X, and the client instance goes to an expecting state.
- If the client instance U^i is in an expecting state, a query Send$(U^i, (S, Y^\star))$ is processed by computing the session key and producing an authenticator. We apply the following rules:
  - ►**Rule U2**$^{(7)}$ – Lookup $(pw, *, \perp, \mathcal{E}, Y^\star) \in \Lambda_\mathcal{E}$. If found, define Encrypt$_7$ as true and abort the game.
  - ►**Rule U3**$^{(7)}$ – Compute the session key $sk_U = \mathcal{H}_2(U\|S\|X\|Y^\star)$ and the authenticator $Auth = \mathcal{H}_3(U\|S\|X\|Y^\star)$.

  Finally the query is answered with $Auth$, the client instance accepts and terminates. Our simulation also adds $((U, X), (S, Y^\star), Auth)$ to Λ_Ψ.

We answer to the Send-queries to the server as follows:

- A Send$(S^j, (U, X))$-query is processed according to the following rule:
  - ►**Rule S1**$^{(7)}$ – Choose a random $Y^\star \in \bar{\mathbb{G}}$. If $(*, Y^\star) \in \Lambda_S$, one aborts the game, otherwise adds the record $(j, Y^\star)$ to Λ_S. Then, compute $Y = \mathcal{D}_{pw}(Y^\star)$.

  Finally, the query is answered with $S, Y^\star$ and the server instance goes to an expecting state.
- If the server instance S^j is in an expecting state, a query Send(S^j, H) is processed according to the following rules:
  - ►**Rule S2**$^{(7)}$ – Check if $(X, Y^\star, H) \in \Lambda_\Psi$. If this is not the case, then reject the authenticator: terminate, without accepting. Check if $(1, U\|S\|X\|Y\|*, H) \in \Lambda_\mathcal{A}$. If this is the case, we define the event Auth$'_7$ to be true, and abort the game.

  If the server instance has not terminated, it accepts and moves on to apply the following rule:
  - ►**Rule S3**$^{(7)}$ – Compute the session key $sk_S = \mathcal{H}_2(U\|S\|X\|Y^\star)$.

  Finally, the server instance terminates.

Fig. 6. Simulation of the Send-queries in $\mathbf{G}_7$

We first rewrite the **Rule U2**:

►**Rule U2-**$^{(7)}$ – Does nothing.

►**Rule U2+**$^{(7)}$ – Lookup $(pw, *, \perp, \mathcal{E}, Y^\star) \in \Lambda_\mathcal{E}$. If found, define Encrypt$_7$ as true (and abort the game).

We then modify the organization of the **Rule S1** and the **Rule S2**:

►**Rule S1-**$^{(7)}$ – Choose a random $Y^\star \in \bar{\mathbb{G}}$. If $(*, Y^\star) \in \Lambda_S$, one aborts the game, otherwise adds the record $(j, Y^\star)$ to Λ_S.

►**Rule S2-**$^{(7)}$ – Check if $((U, X), (S, Y^\star), H) \in \Lambda_\Psi$. If this is not the case, then reject the authenticator: terminate, without accepting.

►**Rule S2+**$^{(7)}$ – Compute $Y = \mathcal{D}_{pw}(Y^\star)$, and lookup $(1, U\|S\|X\|Y\|*, H) \in \Lambda_\mathcal{A}$. If found, define Auth$'_7$ as true (and abort the game).

Fig. 7. Rewriting of some Rules in $\mathbf{G}_7$

A.2 Conclusion of the Proof of Theorem 1

By summing up all the relations, one completes the proof. From Equations (1), (2), (3), (4) and (5),

$$\begin{aligned} |\Pr[\mathsf{S}_4] - \Pr[\mathsf{S}_0]| &\leq \frac{q_{\mathcal{E}}^2}{2(q-1)} + \frac{q_S q_{\mathcal{E}}}{q-1} + \frac{2q_{\mathcal{E}}^2 + q_S^2}{2(q-1)} + \frac{q_h^2}{2^{\ell_1+1}} + \Pr[\mathsf{Encrypt}_4] \\ &\leq \frac{(2q_{\mathcal{E}} + q_S)^2}{2(q-1)} + \frac{q_h^2}{2^{\ell_1+1}} + \Pr[\mathsf{Encrypt}_4] \end{aligned}$$

From Equations (6 – 8), $|\Pr[\mathsf{Encrypt}_7] - \Pr[\mathsf{Encrypt}_4]|$ and $|\Pr[\mathsf{S}_7] - \Pr[\mathsf{S}_4]|$ are both upper-bounded by

$$\frac{q_s}{2^{\ell_1}} + \Pr[\mathsf{Auth}'_6] + \Pr[\mathsf{AskH}_7] \leq \frac{q_s}{2^{\ell_1}} + \Pr[\mathsf{Auth}'_7] + 2\Pr[\mathsf{AskH}_7]. \tag{12}$$

Then,

$$\begin{aligned} |\Pr[\mathsf{S}_7] - \Pr[\mathsf{S}_0]| \leq &\frac{(2q_{\mathcal{E}} + q_S)^2}{2(q-1)} + \frac{q_h^2}{2^{\ell_1+1}} + \frac{2q_s}{2^{\ell_1}} \\ &+ \Pr[\mathsf{Encrypt}_7] + 2\Pr[\mathsf{Auth}'_7] + 4\Pr[\mathsf{AskH}_7]. \end{aligned}$$

From Equations (9), (10) and (11), one gets

$$\Pr[\mathsf{Encrypt}_7] \leq \frac{q_s}{2N} \qquad \Pr[\mathsf{Auth}'_7] \leq \frac{q_s}{2N} \qquad \Pr[\mathsf{AskH}_7] \leq q_h \mathsf{Succ}_{\mathbb{G}}^{\mathsf{cdh}}(t'), \tag{13}$$

which concludes the proof. □

B Proof of Theorem 3

We can actually use the proof presented in Section 3.2, since

$$\mathsf{Adv}_{\mathsf{oeke}}^{\mathsf{c-auth}}(\mathcal{A}) = \Pr[\mathsf{Auth}_0],$$

and see that in game $\mathbf{G}_6$, $\Pr[\mathsf{Auth}_6] = 0$, and Equations (2), (3), (4), (5), (6), and (7) extends to

$$\begin{aligned} |\Pr[\mathsf{Auth}_1] - \Pr[\mathsf{Auth}_0]| &\leq \frac{q_{\mathcal{E}}^2}{2(q-1)} & |\Pr[\mathsf{Auth}_2] - \Pr[\mathsf{Auth}_1]| &\leq \frac{q_S q_{\mathcal{E}}}{q-1} \\ |\Pr[\mathsf{Auth}_3] - \Pr[\mathsf{Auth}_2]| &\leq \frac{2q_{\mathcal{E}}^2 + q_S^2}{2(q-1)} + \frac{q_h^2}{2^{\ell_1+1}} & |\Pr[\mathsf{Auth}_4] - \Pr[\mathsf{Auth}_3]| &\leq \Pr[\mathsf{Encrypt}_4] \\ |\Pr[\mathsf{Auth}_5] - \Pr[\mathsf{Auth}_4]| &\leq \frac{q_s}{2^{\ell_1}} & |\Pr[\mathsf{Auth}_6] - \Pr[\mathsf{Auth}_5]| &\leq \Pr[\mathsf{Auth}'_6]. \end{aligned}$$

Then, using Equations (12) from the conclusion of the previous proof, and Equation (8), one gets,

$$\begin{aligned} \mathsf{Adv}_{\mathsf{oeke}}^{\mathsf{c-auth}}(\mathcal{A}) &\leq \frac{q_{\mathcal{E}}^2}{2(q-1)} + \frac{q_S q_{\mathcal{E}}}{q-1} + \frac{2q_{\mathcal{E}}^2 + q_S^2}{2(q-1)} + \frac{q_h^2}{2^{\ell_1+1}} + \Pr[\mathsf{Encrypt}_4] + \frac{q_s}{2^{\ell_1}} + \Pr[\mathsf{Auth}'_6] \\ &\leq \frac{(2q_{\mathcal{E}} + q_S)^2}{2(q-1)} + \frac{q_h^2 + 2q_s}{2^{\ell_1+1}} \\ &\quad + \left(\Pr[\mathsf{Encrypt}_7] + \frac{q_s}{2^{\ell_1}} + \Pr[\mathsf{Auth}'_7] + 2\Pr[\mathsf{AskH}_7]\right) \\ &\quad + \left(\Pr[\mathsf{Auth}'_7] + \Pr[\mathsf{AskH}_7]\right) \\ &\leq \frac{(2q_{\mathcal{E}} + q_S)^2}{2(q-1)} + \frac{q_h^2 + 4q_s}{2^{\ell_1+1}} + \Pr[\mathsf{Encrypt}_7] + 2\Pr[\mathsf{Auth}'_7] + 3\Pr[\mathsf{AskH}_7], \end{aligned}$$

which concludes the proof, using Equation (13). □

C Security Proof of AuthA

Proving the security of this new protocol follows the same path as the one in Section 3.2, until the Game $\mathbf{G}_8$:

Game $\mathbf{G}_8$***:*** In that game, we simulate the executions using the random self-reducibility of the Diffie-Hellman problem, given one Diffie-Hellman instance (A, B). We first choose a random element $\gamma \in \mathbb{Z}_q^\star$ and define $pw_S = A^\gamma$. We also add the record (γ, pw_S) to Λ_A.

▶**Rule U1**$^{(8)}$ – Choose a random element $\alpha \in \mathbb{Z}_q^\star$, and compute $X = A^\alpha$. Also add the record (α, X) to Λ_A.

▶**Rule** $\mathcal{D}^{(8)}$ – Choose a random element $\beta \in \mathbb{Z}_q^\star$, and compute the answer $Z = B^\beta$. Also add the record (β, Z) to Λ_B. If $(*, Z, *, *, Z^\star) \in \Lambda_\mathcal{E}$, one aborts the game, otherwise adds the record $(k, Z, \perp, \mathcal{D}, Z^\star)$ to $\Lambda_\mathcal{E}$.

$$\Pr[\mathsf{AskH}_8] = \Pr[\mathsf{AskH}_7]. \tag{14}$$

Remember that AskH_8 means that the adversary $\mathcal{A}$ queried the random oracles $\mathcal{H}_0$ or $\mathcal{H}_1$ on $U\|S\|X\|Y\|\mathsf{CDH}(X,Y)$, and thus $\mathcal{H}$ on $U\|S\|X\|Y\|\mathsf{CDH}(X,Y)$ or $*\|\mathsf{CDH}(pw_S, Y)$. By picking randomly in the $\Lambda_\mathcal{A}$-list, with probability $1/q_h$, we can get the Diffie-Hellman secret value. This is a triple $(X, Y, \mathsf{CDH}(X,Y))$. One then simply looks up into Λ_A and Λ_B to get α and β such that $X = A^\alpha$ and $Y = B^\beta$:

$$\mathsf{CDH}(X,Y) = \mathsf{CDH}(A^\alpha, B^\beta) = \mathsf{CDH}(A,B)^{\alpha\beta}.$$

Thus:

$$\Pr[\mathsf{AskH}_8] \le q_h \mathsf{Succ}_\mathbb{G}^{\mathsf{cdh}}(t'). \tag{15}$$

This concluded the proof. □

D Forward-Secrecy

The previous security results and proofs do not deal with forward-secrecy. Considering forward-secrecy requires to take into account a new kind of query that we call the $\mathsf{Corrupt}$-query (any other kinds of queries can still be asked, before but also after this one):

– $\mathsf{Corrupt}(I)$: This query models the attacks resulting in the password pw of this party I to be revealed. $\mathcal{A}$ gets back from its query pw but does not get any internal data of I.

Then we define a new flavor of freshness, saying that an instance is **Fresh** (or holds a **Fresh** key sk) if the following conditions hold. First, the instance has computed and accepted a session key. Second, no $\mathsf{Corrupt}$-query has been made by the adversary since the beginning of the game (before the session key is accepted). Third, neither it nor its partner have been asked for a Reveal-query.

This security level means that the adversary does not learn any information about *previously* established session keys when making a $\mathsf{Corrupt}$-query. We thus denote by $\mathsf{Adv}_P^{\mathsf{ake-fs}}(\mathcal{A})$ the advantage an adversary can get on a fresh key, in the protocol P, with the ability to make a $\mathsf{Corrupt}$-query.

Theorem 4 (AKE-FS Security). *Let us consider the* OEKE *protocol, where* **SK** *is the session-key space and* Password *a finite dictionary of size N equipped with the uniform distribution. Let $\mathcal{A}$ be an adversary against the AKE security of* OEKE *within a time bound t, with less than q_s interactions with the parties and q_p passive eavesdroppings, and, asking q_h hash-queries and q_e encryption/decryption queries. Then we have*

$$\mathsf{Adv}_{\mathsf{oeke}}^{\mathsf{ake-fs}}(\mathcal{A}) \le 3 \times \frac{q_s}{N} + 4q_h(1 + (q_s + q_p)^2) \times \mathsf{Succ}_\mathbb{G}^{\mathsf{cdh}}(t') + \frac{(2q_e + 3q_s + 3q_p)^2}{q-1} + \frac{q_h^2 + 4q_s}{2^{\ell_1}}.$$

where $t' \le t + (q_s + q_p + q_e) \cdot \tau_\mathbb{G}$, with $\tau_\mathbb{G}$ denoting the computational time for an exponentiation in $\mathbb{G}$. (Recall that q is the order of $\mathbb{G}$.)

Proof. To deal with forward-secrecy, we define event $\mathsf{Corrupted}$ as the event that $\mathcal{A}$ asks a $\mathsf{Corrupt}$-query, and we refine events $\mathsf{Encrypt}$, Auth, Auth' and AskH respectively into $\mathsf{EncryptBC}$, AuthBC, AuthBC' and AskHBC respectively:

$$\begin{aligned} \mathsf{EncryptBC}_k &:= \mathsf{Encrypt}_k \prec \mathsf{Corrupted} \qquad & \mathsf{AuthBC}_k &:= \mathsf{Auth}_k \prec \mathsf{Corrupted} \\ \mathsf{AuthBC}'_k &:= \mathsf{Auth}'_k \prec \mathsf{Corrupted} \qquad & \mathsf{AskHBC}'_k &:= \mathsf{AskH}_k \prec \mathsf{Corrupted} \end{aligned}$$

that is $\mathsf{EncryptBC}_k$, AuthBC_k, AuthBC'_k or AskHBC_k respectively occur if $\mathsf{Encrypt}_k$, Auth_k, Auth'_k or AskH_k respectively occur **before** corrupting a player.

We can base the proof on a similar sequence of games as before, but just modifying some rules before any corruption:

►**Rule S2$^{(6)}$** – If $(X, Y^\star, H) \notin \Lambda_\Psi$, and either $\mathsf{Corrupted} = \mathsf{false}$ or ($\mathsf{Corrupted} = \mathsf{true}$ and $(1, U\|S\|X\|Y\|K_S, H) \notin \Lambda_\mathcal{A}$), then reject the authenticator: terminate, without accepting. Moreover, if $\mathsf{Corrupted} = \mathsf{false}$ and $(1, U\|S\|X\|Y\|*, H) \in \Lambda_\mathcal{A}$ we define the event AuthBC'_6 to be true, and abort the game.

►**Rule U3$^{(7)}$** – If $\mathsf{Corrupted} = \mathsf{false}$, then compute the session key $sk_U = \mathcal{H}_2(U\|S\|X\|Y^\star)$ and the authenticator $Auth = \mathcal{H}_3(U\|S\|X\|Y^\star)$. Otherwise, compute the session key $sk_U = \mathcal{H}_0(U\|S\|X\|Y\|K_U)$ and the authenticator $Auth = \mathcal{H}_1(U\|S\|X\|Y\|K_U)$.

►**Rule S3$^{(7)}$** – If $\mathsf{Corrupted} = \mathsf{false}$, then compute the session key $sk_S = \mathcal{H}_2(U\|S\|X\|Y^\star)$. Otherwise, compute the session key $sk_S = \mathcal{H}_0(U\|S\|X\|Y\|K_S)$.

►**Rule U2$^{(7)}$** – Lookup $(pw, *, \perp, \mathcal{E}, Y^\star) \in \Lambda_\mathcal{E}$. If found, define $\mathsf{Encrypt}_7$ as true and abort the game. Otherwise, compute $Y = \mathcal{D}_{pw}(Y^\star)$. If $\mathsf{Corrupted} = \mathsf{false}$, furthermore define $K_U = Y^\theta$.

►**Rule S1$^{(7)}$** – Choose a random $Y^\star \in \bar{\mathbb{G}}$. If $(*, Y^\star) \in \Lambda_S$, one aborts the game, otherwise adds the record $(j, Y^\star)$ to Λ_S. Then, compute $Y = \mathcal{D}_{pw}(Y^\star)$. If $\mathsf{Corrupted} = \mathsf{false}$, furthermore lookup $(pw, Y, \varphi, *, Y^\star) \in \Lambda_\mathcal{E}$ to define φ (we thus have $Y = g^\varphi$), and compute $K_S = X^\varphi$.

By evaluating the events $\mathsf{Encrypt}_7$ and Auth_7 at the corruption time, one gets as before

$$|\Pr[\mathsf{S}_6] - \Pr[\mathsf{S}_0]| \le \frac{(2q_\mathcal{E} + q_S)^2}{2(q-1)} + \frac{q_h^2}{2^{\ell_1+1}} + \Pr[\mathsf{EncryptBC}_4] + \frac{q_s}{2^{\ell_1}} + \Pr[\mathsf{AuthBC}'_6],$$

$$\Pr[\mathsf{EncryptBC}_4] \le \frac{q_s}{N} + \frac{q_s}{2^{\ell_1}} + q_h \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t') \qquad \Pr[\mathsf{AuthBC}'_6] \le \frac{q_s}{2N} + q_h \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t').$$

As a consequence,

$$|\Pr[\mathsf{S}_6] - \Pr[\mathsf{S}_0]| \le \frac{3q_s}{2N} + 2q_h \times \mathsf{Succ}^{\mathsf{cdh}}_{\mathbb{G}}(t') + \frac{(2q_\mathcal{E} + q_S)^2}{2(q-1)} + \frac{q_h^2}{2^{\ell_1+1}} + \frac{2q_s}{2^{\ell_1}}. \qquad (16)$$

We now go back the game $\mathbf{G}_6$, as presented on Figure 8. We furthermore abort the game where the events $\mathsf{EncryptBC}_6$ or AuthBC'_6 happen to be true.

Game* $\mathbf{G}_7$*: We now have to make a different analysis: we need to know the private exponents of (almost) all the instances of the parties, since the adversary may send the authenticator after making the $\mathsf{Corrupt}$-query, and thus knowing the password. Otherwise, a later Reveal-query would not be perfect. Therefore, one first bets on an execution (passive or active) to be tested: one chooses a random index $\mu \in \{1, \ldots, q_s + q_p\}$ and a random index $\nu \in \{1, \ldots, q_s + q_p\}$. If the Test-query does not correspond to the client involved in the μ-th Send-query, and the server involved in the ν-th Send-query, then one aborts the game, outputting a random bit b'. Since the Test-query can only be asked to an instance that has accepted before any corruption and that only simulated keys can be asked,

$$\Pr[\mathsf{S}_7] = \frac{1}{(q_s + q_p)^2} \times \Pr[\mathsf{S}_6] + \left(1 - \frac{1}{(q_s + q_p)^2}\right) \times \frac{1}{2}.$$

We answer to the Send-queries to the client as follows:

- A $\mathsf{Send}(U^i, \mathtt{Start})$-query is processed according to the following rule:
  ►Rule U1$^{(6)}$ – Choose a random exponent $\theta \in \mathbb{Z}_q^\star$ and compute $X = g^\theta$.
  Then the query is answered with U, X, and the client instance goes to an expecting state.
- If the client instance U^i is in an expecting state, a query $\mathsf{Send}(U^i, (S, Y^\star))$ is processed by computing the session key and producing an authenticator. We apply the following rules:
  ►Rule U2$^{(6)}$ – Lookup $(pw, *, \perp, \mathcal{E}, Y^\star) \in \Lambda_\mathcal{E}$. If found, define $\mathsf{Encrypt}_6$ as true. Otherwise, compute $Y = \mathcal{D}_{pw}(Y^\star)$. Furthermore define $K_U = Y^\theta$.
  ►Rule U3$^{(6)}$ – Compute the session key $sk_U = \mathcal{H}_0(U\|S\|X\|Y\|K_U)$ and the authenticator $Auth = \mathcal{H}_1(U\|S\|X\|Y\|K_U)$.
  Finally the query is answered with $Auth$, the client instance accepts and terminates. Our simulation also adds $(X, Y^\star, Auth)$ to Λ_Ψ.

We answer to the Send-queries to the server as follows:

- A $\mathsf{Send}(S^j, (U, X))$-query is processed according to the following rule:
  ►Rule S1$^{(6)}$ – Choose a random $Y^\star \in \bar{\mathbb{G}}$. If $(*, Y^\star) \in \Lambda_S$, one aborts the game, otherwise adds the record $(j, Y^\star)$ to Λ_S. Then, compute $Y = \mathcal{D}_{pw}(Y^\star)$, lookup $(pw, Y, \varphi, *, Y^\star) \in \Lambda_\mathcal{E}$ to define φ (we thus have $Y = g^\varphi$), and compute $K_S = X^\varphi$.
  Finally, the query is answered with $S, Y^\star$ and the server instance goes to an expecting state.
- If the server instance S^j is in an expecting state, a query $\mathsf{Send}(S^j, H)$ is processed according to the following rules:
  ►Rule S2$^{(6)}$ – If $(X, Y^\star, H) \notin \Lambda_\Psi$, and either $\mathsf{Corrupted} = \mathsf{false}$ or ($\mathsf{Corrupted} = \mathsf{true}$ and $(1, U\|S\|X\|Y\|K_S, H) \notin \Lambda_\mathcal{A}$), then reject the authenticator: terminate, without accepting. Moreover, if $(1, U\|S\|X\|Y\|*, H) \in \Lambda_\mathcal{A}$ we define the event Auth'_6 to be true.
  If the server instance has not terminated, it accepts and goes on, applying the following rule:
  ►Rule S3$^{(6)}$ – Compute the session key $sk_S = \mathcal{H}_0(U\|S\|X\|Y\|K_S)$.
  Finally, the server instance terminates.

Fig. 8. Simulation of the Send-queries in $\mathbf{G}_6$

Then,

$$\left|\Pr[\mathsf{S}_6] - \frac{1}{2}\right| = (q_s + q_p)^2 \times \left|\Pr[\mathsf{S}_7] - \frac{1}{2}\right|. \tag{17}$$

Game $\mathbf{G}_8$***:*** We now inject a CDH instance into this specific execution: we are given (A, B), with the discrete logarithms a and b

►Rule U1$^{(8)}$ – If this corresponds to the μ-th instance of the client, set $\theta = a$, otherwise, choose a random element $\theta \in \mathbb{Z}_q^\star$. Then compute $X = g^\theta$.

►Rule $\mathcal{D}^{(8)}$ – If this corresponds to the ν-th instance of the server, set $\varphi = b$, otherwise choose a random element $\varphi \in \mathbb{Z}_q^\star$. Then compute $Z = B^\varphi$. If $(*, Z, *, *, Z^\star) \in \Lambda_\mathcal{E}$, one aborts the game. One finally adds the record $(k, Z, \varphi, \mathcal{D}, Z^\star)$ to $\Lambda_\mathcal{E}$.

The games $\mathbf{G}_8$ and $\mathbf{G}_7$ are perfectly indistinguishable:

$$\Pr[\mathsf{S}_7] = \Pr[\mathsf{S}_8]. \tag{18}$$

Game $\mathbf{G}_9$***:*** In that game, the session key and the authenticator of this specific execution of the protocol is defined using private random oracles $\mathcal{H}_2$ and $\mathcal{H}_3$, independent from $\mathcal{H}_0$ and $\mathcal{H}_1$. For that, we modify the following rules:

►Rule U2$^{(9)}$ – Lookup $(pw, *, \perp, \mathcal{E}, Y^\star) \in \Lambda_\mathcal{E}$. If found, define $\mathsf{Encrypt}_9$ as true. If this does not correspond to the μ-th instance of the client, one computes $Y = \mathcal{D}_{pw}(Y^\star)$ and defines $K_U = Y^\theta$ (otherwise we won't need it).

►Rule U3$^{(9)}$ – If this corresponds to the μ-th instance of the client, one computes the session key $sk_U = \mathcal{H}_2(U\|S\|X\|Y^\star)$ and the authenticator $Auth = \mathcal{H}_3(U\|S\|X\|Y^\star)$. Otherwise, compute the session key $sk_U = \mathcal{H}_0(U\|S\|X\|Y\|K_U)$ and the authenticator $Auth = \mathcal{H}_1(U\|S\|X\|Y\|K_U)$.

- ►**Rule S1**$^{(9)}$ – Choose a random $Y^\star \in \bar{\mathbb{G}}$. If $(*, Y^\star) \in \Lambda_S$, one aborts the game, otherwise adds the record $(j, Y^\star)$ to Λ_S. If this does not correspond to the ν-th instance of the server, one computes $Y = \mathcal{D}_{pw}(Y^\star)$, looks up $(pw, Y, \varphi, *, Y^\star) \in \Lambda_{\mathcal{E}}$ to define φ (we thus have $Y = g^\varphi$), and computes $K_S = X^\varphi$ (otherwise we won't need it).

- ►**Rule S3**$^{(9)}$ – If this corresponds to the ν-th instance of the server, one computes the session key $sk_S = \mathcal{H}_2(U\|S\|X\|Y^\star)$. and the authenticator $Auth = \mathcal{H}_3(U\|S\|X\|Y^\star)$. Otherwise, compute the session key $sk_U = \mathcal{H}_0(U\|S\|X\|Y\|K_S)$ and the authenticator $Auth = \mathcal{H}_1(U\|S\|X\|Y\|K_S)$.

The games $\mathbf{G}_9$ and $\mathbf{G}_8$ are indistinguishable unless the following event AskH_9 occurs: $\mathcal{A}$ queries the hash functions $\mathcal{H}_0$ or $\mathcal{H}_1$ on $U\|S\|X\|Y\|\mathsf{CDH}(X,Y)$:

$$|\Pr[\mathsf{S}_9] - \Pr[\mathsf{S}_8]| \leq \Pr[\mathsf{AskH}_9]. \tag{19}$$

Game $\mathbf{G}_{10}$***:*** Now, we are not given the discrete logarithms a and b anymore:

- ►**Rule U1**$^{(10)}$ – If this corresponds to the μ-th instance of the client, set $X = A$, otherwise, choose a random element $\theta \in \mathbb{Z}_q^\star$ and compute $X = g^\theta$.

- ►**Rule** $\mathcal{D}^{(10)}$ – If this corresponds to the ν-th instance of the server, set $Z = B$ and $\varphi = \perp$, otherwise choose a random element $\varphi \in \mathbb{Z}_q^\star$ and compute $Z = B^\varphi$. Finally, if $(*, Z, *, *, Z^\star) \in \Lambda_{\mathcal{E}}$, one aborts the game. One then adds the record $(k, Z, \varphi, \mathcal{D}, Z^\star)$ to $\Lambda_{\mathcal{E}}$.

Since K_U and K_S are not required for this execution of the protocol (the session key and the authenticator are defined using independent private random oracles on X and $Y^\star$ only), the two games are indistinguishable:

$$\Pr[\mathsf{S}_9] = \Pr[\mathsf{S}_{10}] \qquad \Pr[\mathsf{AskH}_9] = \Pr[\mathsf{AskH}_{10}]. \tag{20}$$

Furthermore, it is now clear that

$$\Pr[\mathsf{AskH}_{10}] = q_h \times \mathsf{Succ}_{\mathbb{G}}^{\mathsf{cdh}}(t'). \tag{21}$$

As a conclusion, from the Equations (16), (17), (18), (19), (20) and (21),

$$\left|\Pr[\mathsf{S}_6] - \frac{1}{2}\right| \leq 2(q_s + q_p)^2 \times \Pr[\mathsf{AskH}_9] \leq 2(q_s + q_p)^2 q_h \times \mathsf{Succ}_{\mathbb{G}}^{\mathsf{cdh}}(t').$$

This security result can definitely be improved using the random self-reducibility of the Diffie-Hellman problem. Namely, one could remove the factor $(q_s + q_p)^2$, but this would make the reduction much more intricate. □

This extended abstract appeared in
Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA '08)
April 8–11, 2008, 2008, San Francisco, CA, USA – T. Malkin Ed. Springer-Verlag, LNCS 4964, pages 335–351.

Efficient Two-Party Password-Based Key Exchange Protocols in the UC framework

Michel Abdalla[1], Dario Catalano[2], Céline Chevalier[1], and David Pointcheval[1]

[1] École Normale Supérieure, LIENS-CNRS-INRIA, Paris, France
[2] Universit di Catania, Italy

Abstract. Most of the existing password-based authenticated key exchange protocols have proofs either in the indistinguishability-based security model of Bellare, Pointcheval, and Rogaway (BPR) or in the simulation-based of Boyko, MacKenzie, and Patel (BMP). Though these models provide a security level that is sufficient for most applications, they fail to consider some realistic scenarios such as participants running the protocol with different but possibly related passwords. To overcome these deficiencies, Canetti *et al.* proposed a new security model in the universal composability (UC) framework which makes no assumption on the distribution on passwords used by the protocol participants. They also proposed a new protocol, but, unfortunately, the latter is not as efficient as some of the existing protocols in BPR and BMP models. In this paper, we investigate whether some of the existing protocols that were proven secure in BPR and BMP models can also be proven secure in the new UC model and we answer this question in the affirmative. More precisely, we show that the protocol by Bresson, Chevassut, and Pointcheval (BCP) in CCS 2003 is also secure in the new UC model. The proof of security relies in the random-oracle and ideal-cipher models and works even in the presence of adaptive adversaries, capable of corrupting players at any time and learning their internal states.

1 Introduction

Password-based authenticated key exchange (PAKE) protocols allow users to securely establish a common key over an insecure channel only using a low-entropy, human-memorizable, secret key called a password. Since PAKE protocols do not require complex public-key infrastructure (PKI) or trusted hardware capable of storing high-entropy keys, they have become quite popular since being introduced by Bellovin and Merritt [3].

Due to the low entropy of passwords, PAKE protocols are subject to dictionary attacks in which the adversary tries to break the security of the scheme by trying all values for the password in the small set of the possible values (i.e., the dictionary). Unfortunately, these attacks can be quite damaging since the attacker has a non-negligible probability of succeeding. To address this problem, one should invalidate or block the use of a password whenever a certain number of failed attempts occurs. However, this is only effective in the case of *online* dictionary attacks in which the adversary must be present and interact with the system in order to be able to verify whether its guess is correct. Thus, the goal of PAKE protocol is restrict the adversary to *online* dictionary attacks only. In other words, *off-line* dictionary attacks, in which the adversary verifies if a password guess is correct without interacting with the system, should not be possible in a PAKE protocol.

Security Models. Even though the notion of password-based authentication dates back to the seminal work by Bellovin and Merritt [3], it took several years for the first formal security models to appear in the literature [5, 4]. In [5], Bellare, Pointcheval, and Rogaway (BPR) proposed an indistinguishability-based security model extending the framework of Bellare and Rogaway [7, 8] while, in [4], Boyko, MacKenzie, and Patel (BMP) proposed a simulation-based security model based on the framework of Shoup [18]. In both cases, the level of security provided by the models is quite reasonable and sufficient for most applications and it captures the intuition given above in which the success of an adversary in breaking the security of a scheme should be limited to its online attempts.

Unfortunately, as pointed out by Canetti *et al.* [10], the BPR and BMP security models are not as general or as strong as they could be and they fail to consider some realistic scenarios such as participants running the protocol with different but possibly related passwords. To overcome these deficiencies, Canetti *et al.* [10] proposed a new security model for PAKE schemes in the universal composability (UC) framework [9] which makes no assumption on the distribution on passwords used by the protocol participants. Their model was later extended to the verifier-based scenario by Gentry *et al.* [13].

In addition to the new security model, Canetti *et al.* [10] also proposed a new protocol based on the PAKE schemes by Katz, Ostrovsky, and Yung [15] and by Gennaro and Lindell schemes [12] and proved it secure in the new model against static adversaries based on standard computational assumptions. Unfortunately, the new protocol is not as efficient as some of the existing protocols in BPR and BMP models (e.g., [2, 1, 15, 17]), an issue that can significantly limit its applicability. Given this limitation, one natural question to ask is whether some of the more efficient protocols that were proven secure in BPR and BMP models can also be proven secure in the model of Canetti *et al.* [10]. In this paper, we answer this question in the affirmative by showing that the protocol by Bresson, Chevassut, and Pointcheval (BCP) [2] is also secure in the model of Canetti *et al.* [10]. We view this as the main contribution of our paper.

In addition to proving the security of the BCP protocol in the model of Canetti *et al.* [10], another contribution of our paper is to show that their protocol remains secure even against adaptive adversaries, capable of corrupting adversaries at any time and learning their internal states. Despite this being first time that such a strong security level is achieved in the password-based scenario, we do not consider this result very surprising given the use of the random-oracle and ideal-cipher models in the security proof.

Organization. In Section 2, we extend the ideal functionality of PAKE protocols to include client authentication, which not only ensures the parties that nobody else knows the common secret, but also that they actually share the same secret. As in [10], passwords are chosen by the environment who then hands them to the parties as input. This is the strongest security model, since it does not assume any distribution on passwords. Furthermore, it allows the environment to even make players run the protocol with different (possibly related) passwords. For example, this models a user mistyping a password. As in [10], we also provide the adversary with a Test-Password query to model the vulnerability of the passwords (whose entropy may be low). This models the case in which the adversary tries to impersonate a player by guessing its password. If the guess is correct (which may happen with non-negligible probability), the adversary should succeed in its impersonation.

Next, in Section 3, we recall the password-based protocol of [2] and prove it secure in the new extended model, even against adaptive adversaries which can perform strong corruptions at any time. The proof is given in Section 4. As we mentioned above, this is the first time that such a strong security level is achieved in the password-based scenario: adaptive and strong corruptions in the UC framework.

In the appendix, we also provide ideal functionalities for the ideal-cipher and the random-oracle models [6].

2 Definition of Security

Notations. We denote by k the security parameter. An event is said to be negligible if it happens with probability that is less than the inverse of any polynomial in k. If G is a finite set, $x \overset{R}{\leftarrow} G$ indicates the process of selecting x uniformly and at random in G (thus we implicitly assume that G can be sampled efficiently).

The UC Framework. Throughout this paper we assume basic familiarity with the universal composability framework. Here we provide a brief overview of the framework. The interested reader is referred to [9] for complete details. In a nutshell, security in the UC framework is defined in terms of an ideal functionality $\mathcal{F}$, which is basically a trusted party that interacts with a set of players to compute some given function f. In particular, the players hand their input to $\mathcal{F}$ which computes f on the received inputs and gives back to each player the appropriate output. Thus, in this idealized setting, security is inherently guaranteed, as any adversary, controlling some of the parties, can only learn (and possibly modify) the data of corrupted players. In order to prove that a candidate protocol

π realizes the ideal functionality, one considers an environment $\mathcal{Z}$, which is allowed to provide inputs to all the participants and that aims to distinguish the case where it receives the outputs produced from a real execution of the protocol (involving all the parties and an adversary $\mathcal{A}$, controlling some of the parties and the communication among them), from the case where it receives outputs obtained from an ideal execution of the protocol (involving only dummy parties interacting with $\mathcal{F}$ and an ideal adversary $\mathcal{S}$ also interacting with $\mathcal{F}$). Then we say that π realizes the functionality $\mathcal{F}$ if for every (polynomially bounded) $\mathcal{A}$, there exists a (polynomially bounded) $\mathcal{S}$ such that no (polynomially bounded) $\mathcal{Z}$ can distinguish a real execution of the protocol from an ideal one with a significant advantage. In particular, the universal composability theorem assures us that π continues to behave like the ideal functionality even if it is executed in an arbitrary network environment.

Session ID's and Player's IDs. In the UC framework there may be many copies of the ideal functionality running in parallel. Each one of such copies is supposed to have a unique session identifier (SID). Every time a message has to be sent to a specific copy of $\mathcal{F}$, such a message should contain the SID of the copy it is intended for. Following [10], we decided to make things simple and to assume that each protocol that realizes $\mathcal{F}$ expects to receive inputs that already contain the appropriate SID. See [10] for further details about this. Moreover we assume that every player starts a new session of the protocol with input ($\texttt{NewSession}$, sid, P_i, P_j, pw, role), where P_i is the identity of the player, pw his or her password, P_j the identity of the player with whom he or she intends to share a session key and role being either client or server.

UC With Joint State. The original UC theorem allows to analyze the security of a system viewed as a single unit, but it says nothing if different protocols share some amount of state and randomness (such as a common reference string, for instance). Thus for the application we have in mind, the UC theorem cannot be used as it is, since different sessions of the protocol share the same random oracles and the same ideal cipher.

In [11] Canetti and Rabin introduced the notion of universal composability with joint state. Informally, they put forward a new composition operation that allows different protocols to have some common state, while preserving security. Very informally, this is done by defining a multisession extension $\hat{\mathcal{F}}$ of $\mathcal{F}$, which basically runs multiple executions of $\mathcal{F}$. Each copy of $\mathcal{F}$ is identified by means of a sub-session id (SSID). This means that, if $\hat{\mathcal{F}}$ receives a message m with SSID $ssid$ it hands m to the copy of $\mathcal{F}$ having SSID $ssid$. If no such copy exists, $\hat{\mathcal{F}}$ invokes a new one on the spot. Notice that, whenever $\hat{\mathcal{F}}$ is executed, the calling protocol has to specify both the SID (i.e. the usual session id, as in any ideal functionality) *and* the SSID.

Adaptive Adversaries. In this paper, we will consider protocols that are secure against adaptive adversaries, i.e. adversaries that are allowed to arbitrarily corrupt players at any moment during the execution of the protocol. The adversary corrupts a player by getting complete access to its internal memory. Note that at the end of an execution of the protocol, the adversary recovers nothing, as if the internal state has been completely erased. In a real execution of the protocol this is modeled by letting the adversary $\mathcal{A}$ obtain the password and the internal state of the corrupted player. Moreover, the adversary can arbitrarily modify the player's strategy. In an ideal execution of the protocol, the simulator $\mathcal{S}$ gets the player's password and has to simulate its internal state, in a way that remains consistent to what already provided to the environment.

The Random Oracle and the Ideal Cipher For lack of space, a description of these functionalities is given in Appendix A.

The Password-Based Key-Exchange Functionality With Client Authentication. In this section, we motivate and present our formulation of an ideal functionality for password-based key exchange with client authentication (see Figure 1). The starting point for our approach is the definition for universally composable password-based key exchange with no authentication [10]. Our aim is to define a functionality that achieves the same effect, except that we also incorporate the authentication of the client. Mutual authentication would have been easier to model. However, client-authentication is usually enough in most cases and often results in more efficient protocols.

First notice that the functionality is not in charge of providing the password(s) to the participants (the client Alice and the server Bob). Rather we let the environment do this. As already pointed out in [10], such an approach allows to model, for example, the case where some users may use the same password for different protocols and, more generally, the case where password(s) are chosen according to some arbitrary distribution (i.e. not necessarily the uniform one). Moreover, notice that allowing the environment to choose the password(s) guarantees forward secrecy, basically for free.

The queries NewSession and TestPwd are dealt with in the same manner as in [10], but we introduce the client authentication in the way the functionality answers the NewKey queries. In the definition of $\mathcal{F}_{pwKE}^{CA}$, the server receives an error if the players don't meet all the conditions to receive the same, randomly-chosen key. We could have chosen to send to the server a pair consisting of a key chosen independently from that of the client and a flag warning the server that the protocol has failed, but we preferred to keep the functionality as straightforward as possible.

Client Authentication. The first reason why the initial functionality didn't achieve this property is that we had to deal with the order of the queries NewKey. More precisely, if the server asks the first query, it is impossible to answer it, because we don't know what is going to happen to the client afterwards: If the session was fresh for both players and the server was the only one to have received his key, the client's session could possibly become compromised or interrupted after the server had received his key, whereas the functionality should have been able to determine whether or not the server should receive a key or an error message. We solved this issue by making it mandatory for the adversary to ask the query for the server after the corresponding query for the client. This is not a strong restriction, since this situation frequently happens in real protocols, and in particular in the one that we are studying: the server has to accept the client before generating the session key.

Thus, if the adversary asks for the key of a client, everything is as before, except that we also provide a flag ready for the session. The aim of this flag is to help determine, when the adversary asks for the key of the server, that the corresponding client has already got her key.

On the other hand, if the adversary asks for the key of a server, the server is given an error message in the easy failure cases (interrupted or compromised sessions, corrupted players – if the passwords are different in the two latter cases). If the session is fresh and the corresponding client hasn't yet received her key, we simply postpone the query of the adversary until the client has received her key. In the latter case, when the client has received her key, the server is given the same key if they have the same password and an error message otherwise. We finally obtain the following definition, which remains trivially secure and correct.

3 Our Scheme

3.1 Description of the Protocol

The protocol presented in Figure 2 is based on that of [2], with two slight differences: In the standard model using the security definition of Bellare *et al.* [5], the session identifier is obtained at the end of the program execution as the concatenation of the random values sent by the players; in particular, it is unique. In contrast, in the model of universal composability [9], these identifiers are uniquely determined in advance, before the beginning of the protocol. Thus, this difference must be taken care of in the definition of the protocol. Another difference has been made, in order to match the definition of the functionality: in case of a failure, the server receives an error message, this feature guaranteeing the client authentication.

3.2 Security Theorem

We consider here the Theorem of Universal Composability in its joint-state version. Let $\widehat{\mathcal{F}}_{pwKE}^{CA}$ be the multi-session extension of $\mathcal{F}_{pwKE}^{CA}$ and let $\mathcal{F}_{RO}$ and $\mathcal{F}_{IC}$ be the ideal functionalities that provide a random oracle and an ideal cipher to all parties. Note that only these two functionalities belong to the joint state.

- $\mathcal{F}_{pwKE}^{CA}$ owns a list L initially empty of values of the form (P_i, P_j, pw).
- **Upon receiving a query $(\mathsf{NewSession}, ssid, P_i, P_j, pw, \mathsf{role})$ from P_i:**
  - Send $(\mathsf{NewSession}, ssid, P_i, P_j, \mathsf{role})$ to $\mathcal{S}$.
  - If this is the first NewSession query, or if it is the second NewSession query and there is a record $(P_j, P_i, pw', \overline{\mathsf{role}}) \in L$, then record $(P_i, P_j, pw, \mathsf{role})$ in L and mark this record fresh.
- **Upon receiving a query $(\mathsf{TestPwd}, ssid, P_i, pw')$ from the adversary $\mathcal{S}$:**
  If there exists a record of the form $(P_i, P_j, pw, \mathsf{role}) \in L$ which is fresh, then do:
  - If $pw = pw'$, mark the record compromised and reply to $\mathcal{S}$ with "correct guess".
  - If $pw \neq pw'$, mark the record interrupted and reply to $\mathcal{S}$ with "wrong guess".
- **Upon receiving a query $(\mathsf{NewKey}, ssid, P_i, sk)$ from $\mathcal{S}$, where $|sk| = k$:**
  If there is a record of the form $(P_i, P_j, pw, \mathsf{role}) \in L$, and this is the first NewKey query for P_i, then:
  If role=client:
  - If the session is compromised, or if one of the two players P_i or P_j is corrupted, then send $(ssid, sk)$ to P_i, record $(P_i, P_j, pw, \mathsf{client}, \mathsf{completed})$ in L, as well as $(ssid, P_i, pw, sk, \mathsf{client}, \mathsf{status}, \mathsf{ready})$ (with status being the status of the session at that moment).
  - Else, if the session is fresh or interrupted, choose a random key sk' whose length is k and send $(ssid, sk')$ to P_i. Record $(P_i, P_j, pw, \mathsf{client}, \mathsf{completed})$ in L, as well as $(ssid, P_i, pw, sk', \mathsf{client}, \mathsf{status}, \mathsf{ready})$ where status stands for fresh or interrupted;

  If role=server:
  - If the session is compromised, if one of the two players P_i or P_j is corrupted, and if there are two records of the form $(P_i, P_j, pw, \mathsf{server})$ and $(P_j, P_i, pw, \mathsf{client})$, set $s = sk$. Otherwise, if the session is fresh and there exists any recorded element of the form $(ssid, P_j, pw', sk', \mathsf{client}, \mathsf{fresh}, \mathsf{ready})$, set $s = sk'$.
    - * If $pw = pw'$, send $(ssid, s)$ to P_i record $(P_i, P_j, pw, \mathsf{server}, \mathsf{completed})$ in L, as well as $(ssid, P_i, pw, s, \mathsf{server}, \mathsf{status})$.
    - * If $pw \neq pw'$, send $(ssid, \mathsf{error})$ to P_i, record $(P_i, P_j, pw, \mathsf{server}, \mathsf{completed})$ in L, as well as $(ssid, P_i, pw, \mathsf{server}, \mathsf{error}, \mathsf{status})$.
  - If the session is fresh and there doesn't exist any recorded element of the form $(ssid, P_j, pw', sk', \mathsf{client}, \mathsf{fresh}, \mathsf{ready})$, then do not do anything;
  - If the session is interrupted, then send $(ssid, \mathsf{error})$ to player P_i, and record in L $(P_i, P_j, pw, \mathsf{server}, \mathsf{completed})$ and $(sid, P_i, pw, \mathsf{server}, \mathsf{error}, \mathsf{interrupted})$.

Fig. 1. Functionality $\mathcal{F}_{pwKE}^{CA}$: it is parametrized by a security parameter k. It interacts with an adversary $\mathcal{S}$ and a set of parties P_1,...,P_n.

Theorem 1 *The above protocol securely realizes $\hat{\mathcal{F}}_{pwKE}^{CA}$ in the $(\mathcal{F}_{RO}, \mathcal{F}_{IC})$-hybrid model, in the presence of adaptive adversaries.*

Client U		Server S
$x \xleftarrow{R} [\![1\,;q-1]\!]$		$y \xleftarrow{R} [\![1\,;q-1]\!]$
$(U1)\ X \leftarrow g^x$	$\xrightarrow{U,X}$	
		$(S2)\ Y \leftarrow g^y$ $Y^* \leftarrow \mathcal{E}_{ssid\Vert pw}(Y)$
	$\xleftarrow{S,Y^*}$	$K_S \leftarrow X^y$
$(U3)\ Y = \mathcal{D}_{ssid\Vert pw}(Y^*)$ $K_U \leftarrow Y^x$ $Auth \leftarrow \mathcal{H}_1(ssid\Vert U\Vert S\Vert X\Vert Y\Vert K_U)$ $sk_U \leftarrow \mathcal{H}_0(ssid\Vert U\Vert S\Vert X\Vert Y\Vert K_U)$ completed	$\xrightarrow{Auth}$	
		$(S4)$ if $(Auth = \mathcal{H}_1(ssid\Vert U\Vert S\Vert X\Vert Y\Vert K_S))$ then $sk_S \leftarrow \mathcal{H}_0(ssid\Vert U\Vert S\Vert X\Vert Y\Vert K_S)$ completed else error

Fig. 2. Client-authenticated two-party password-based key exchange

4 Proof of Theorem 1

4.1 Description of the Proof

In order to show that the protocol UC-realizes the functionality $\mathcal{F}_{pwKE}^{CA}$, we need to show that for all environments and all adversaries, we can construct a simulator such that the interactions, from the one hand between the environment, the players (say, Alice and Bob) and the adversary (the real world), and from the other hand between the environment, the ideal functionality and the simulator (the ideal world), are indistinguishable for the environment.

In this proof, we incrementally define a sequence of games starting with the real execution of the protocol and ending up with game $\mathbf{G}_6$, which we prove to be indistinguishable from the ideal experiment.

Since we have to deal with adaptive corruptions, we consider different cases according to the number of corruptions that have occurred up to now. $\mathbf{G}_0$ is the real world. In $\mathbf{G}_1$, we start by explaining how $\mathcal{S}$ simulates the ideal cipher and the random oracle. Then, in $\mathbf{G}_2$, we get rid of such a situation in which the adversary wins by chance. The passive case, in which no corruption occurs before the end of the protocol, is dealt with in $\mathbf{G}_3$. Next, we completely explain the simulation of the client in $\mathbf{G}_4$, whatever corruption may occur. As for the server, we divide it into two steps: We first show in $\mathbf{G}_5$ how to simulate the last step of the protocol, and then we simulate it from the beginning in $\mathbf{G}_6$. $\mathbf{G}_7$ sums up the situation, and is shown to be indistinguishable from the ideal world.

Note that these games are sequential and built on each other. When we say that a game consider a specific case, one has to understand that in all other cases, the simulation is dealt with as described in the former game.

We first describe two hybrid queries that are going to be used in the games. The GoodPwd query checks whether the password of a certain player is the one we have in mind or not. The SamePwd query checks if the players share the same password, without disclosing it. In some games the simulator has actually full access to the players. In such a case, a $\mathsf{GoodPwd}$ (or a $\mathsf{SamePwd}$) can easily be implemented by simply letting the simulator look at the passwords. When the players are entirely simulated, $\mathcal{S}$ will replace the queries above with a $\mathsf{TestPwd}$ and with a NewKey, respectively.

We say that a flow is *oracle-generated* if it was sent by an honest player and arrives without any alteration to the player it was meant to. We say it is *non-oracle-generated* otherwise, that is either if it was sent by an honest player and modified by the adversary, or if it was sent by a corrupted player or a player impersonated by the adversary.

4.2 Proof of Indistinguishability

Game $\mathbf{G}_0$: Real Game. $\mathbf{G}_0$ is the real game in the random-oracle and ideal-cipher models.

Game $\mathbf{G}_1$: Simulation of the oracles. Here we modify the previous game by simulating the hash and the encryption/decryption oracles, in a quite natural and usual way.

For the ideal cipher, we allow the simulator to maintain a list Λ_ε of entries (queries, responses) of length $q_\varepsilon + q_\mathcal{D}$. Such a list is used by $\mathcal{S}$ to be able to provide answers which are consistent with the following requirements. First, if the simulator receives twice the same question for the same password, it has to give twice the same answer. Second, the simulator should make sure that the simulated scheme (for each password) is actually a permutation. Third, in order to help the simulator to later extract the password used in the encryption of Y^* in the first flow, there should not be two entries (question, answer) with identical ciphertext, but different passwords. More precisely, Λ_ε is actually composed of two sublists: $\Lambda_\varepsilon = \{(ssid, pw, Y, \alpha, \mathcal{E}, Y^*)\} \cup \{(ssid, pw, Y, \alpha, \mathcal{D}, Y^*)\}$. The first (resp. second) sublist is used to indicate that the element Y (resp. Y^*) has been encrypted ("$\mathcal{E}$") (resp. decrypted ("$\mathcal{D}$")) to produce the ciphertext Y^* (resp. Y) via a symmetric encryption algorithm that uses the key $ssid\|pw$. The role of α will be explained below. The simulator manages the list through the following rules:

- For an encryption query $\mathcal{E}_{ssid\|pw}(Y)$ such that $(ssid, pw, Y, *, *, Y^*)$ appears in Λ_ε, the answer is Y^*. Otherwise, choose a random element $Y^* \in G^* = G \setminus \{1\}$. If a record $(*, *, *, *, *, Y^*)$ already belongs to the list Λ_ε, then abort, else add $(ssid, pw, Y, \perp, \mathcal{E}, Y^*)$ to the list.
- For a decryption query $\mathcal{D}_{ssid\|pw}(Y^*)$ such that $(*, pw, Y, *, *, Y^*)$ appears in Λ_ε, the answer is Y. Otherwise, choose a random element $\varphi \in \mathbb{Z}_q{}^*$ and evaluate the answer $Y = g^\varphi$. If $(*, *, Y, *, *, *)$ already belongs to the list Λ_ε, abort, else add $(ssid, pw, Y, \varphi, \mathcal{D}, Y^*)$ to the list.

The two abort-cases will be useful later in the proof: when one sees a ciphertext Y^*, it cannot have been obtained as the encryption with two different passwords, but a unique one.

In addition, the simulator maintains a list $\Lambda_\mathcal{H}$ of length q_h. This list is used to properly manage the queries for the random oracles $\mathcal{H}_0$ and $\mathcal{H}_1$. In particular, the simulator updates $\Lambda_\mathcal{H}$ using the following general rule (n stands for 0 or 1).

- For a hash query $\mathcal{H}_n(q)$ such that (n, q, r) appears in $\Lambda_\mathcal{H}$, the answer is r. Otherwise, choose a random $r \in \{0,1\}^{\ell_{\mathcal{H}_n}}$. If $(n, *, r)$ already belongs to the list $\Lambda_\mathcal{H}$, abort, else add (n, q, r) to the list.

Due to the birthday paradox, $\mathbf{G}_1$ is indistinguishable from the real game $\mathbf{G}_0$.

Game $\mathbf{G}_2$: Case where the adversary wins by chance. This game is almost the same as the previous one. The only difference is that we allow the simulator to abort if the adversary manages to guess *Auth* without having asked a corresponding query to the oracle. This happens with negligible probability so that $\mathbf{G}_2$ and $\mathbf{G}_1$ are indistinguishable.

Game $\mathbf{G}_3$: Passive Case: No Corruption Before Step 4. In this game, we deal with the passive case in which no corruption occurs before step 4. We give the simulator some partial control on the players involved in the protocol. In particular, we assume that the simulator is given oracle access to each player, for the first three rounds of the protocol. Then in $S4$, if no corruption occurred, we require $\mathcal{S}$ to completely simulate their behavior. More precisely, during this game, we consider two cases. If no corruption occurred before $S4$, we require $\mathcal{S}$ to simulate the execution of the protocol on behalf of the two players. If, on the other hand, some party has already been corrupted before starting $S4$, the simulator does nothing. Notice that, in any case, we still allow $\mathcal{S}$ to know the passwords of both players.

If at the beginning of $S4$, the two players are still honest and all the flows were oracle-generated, the simulator asks a SamePwd query. Notice that, since we are assuming that $\mathcal{S}$ knows both passwords, this boils down to verify that both passwords are actually the same.

Now we distinguish two cases. If the two passwords are the same, $\mathcal{S}$ chooses a random key K (in the key space) and "gives" K to all players. Otherwise, $\mathcal{S}$ chooses a random key and gives it to the client whereas the server just receives an error message.

Notice that, if the two players have the same password, such a strategy makes this game indistinguishable with respect to previous one. If, conversely, the players do not have the same passwords, an execution of the protocol in this game is indistinguishable from a real execution except for the risk of collision, which is negligible. This is because, if the two players do not share the same passwords, the server will end-up computing a different $Auth$, thus getting an error message, with all but negligible probability. Hence $\mathbf{G}_3$ and $\mathbf{G}_2$ are indistinguishable.

Game $\mathbf{G}_4$: Simulation of the Client From the Beginning of the Protocol. In this game, we let $\mathcal{S}$ simulate the non-corrupted client from the beginning of the protocol, but we don't allow him to have access to her password anymore. The simulation is done as follows. In $S1$, the client chooses a random x and sends the corresponding X to the server. In $S3$, if she is still honest, then she doesn't ask a decryption query for Y^*.

If all flows were oracle-generated, then she computes $Auth$ with the oracle $\mathcal{H}'_1$ private to the simulator: $Auth = \mathcal{H}'_1(ssid\|U\|S\|X\|Y^*)$ instead of $\mathcal{H}_1$. A problem can occur if the server gets corrupted, as we describe it more formally later on.

Otherwise, if the flow received by the client is not oracle-generated, we face two different cases:

- If the server was corrupted sooner in the protocol, the simulator knows his password, or if the Y^* sent by the adversary in S2 has been obtained via an encryption query, then the simulator recovers his password too (with the help of the encryption list). Then, when receiving Y^*, the client asks a GoodPwd query for the functionality. If it is a correct guess, then $\mathcal{S}$ uses $\mathcal{H}_1$ for the client, otherwise it uses its private oracle $\mathcal{H}'_1$: $Auth = \mathcal{H}'_1(ssid\|U\|S\|X\|Y^*)$.
- If the adversary has not obtained Y^* via an encryption query, there is a negligible chance that it knows the corresponding y and the client also uses $\mathcal{H}'_1$ in this case. The event AskH can then make the game to abort (we will bound its probability later on; simply note that it is negligible and related to the CDH):

  AskH: $\mathcal{A}$ queries one of the oracles $\mathcal{H}_0$ or $\mathcal{H}_1$ on $ssid\|U\|S\|X\|Y\|K_U$ or $ssid\|U\|S\|X\|Y\|K_S$, ie the common value of $ssid\|U\|S\|X\|Y\|CDH(X,Y)$

We now show how to simulate the second part of U3 (the computation of sk_U). We need to separate the cases in which the client remains honest, and those in which she gets corrupted.

- If the client remains honest, she is given sk_U by a query to $\mathcal{H}'_0$ if $Auth$ was obtained by a query to $\mathcal{H}'_1$ and no corruption occurred, and by a query to $\mathcal{H}_0$ if $\mathcal{A}uth$ was obtained by a query to $\mathcal{H}_1$ or if $\mathcal{A}uth$ was obtained by a query to $\mathcal{H}'_1$ and there was a corruption afterwards.
- If she is corrupted during U3, $\mathcal{A}$ is given her internal state: the simulator already knows x and learns her password; it is thus able to compute a correct Y. $\mathcal{S}$ then recomputes $\mathcal{A}uth$ by a query to $\mathcal{H}_1$ (there is no need that this query gives the same value as the value previously computed by the query to $\mathcal{H}'_1$ since $\mathcal{A}uth$ has not been published) and the client is given sk by a query to $\mathcal{H}_0$.

If the two players are honest at the beginning of S4 and all the flows were oracle-generated, there will be no problem as in the former game we prevented the server from computing $Auth$. If the server gets corrupted after $Auth$ has been sent, and if the passwords are the same, the simulator reprograms the oracles such that on the one hand $\mathcal{H}_1(ssid\|U\|S\|X\|Y\|K_U) = \mathcal{H}'_1(ssid\|U\|S\|X\|Y^*)$ and on the other hand $\mathcal{H}_0(ssid\|U\|S\|X\|Y\|K_U) = \mathcal{H}'_0(ssid\|U\|S\|X\|Y^*)$. This programming will only fail if this query to $\mathcal{H}_1$ or $\mathcal{H}_0$ has already been asked before the corruption, in which case the event AskH has happened.

Finally, if the client is being corrupted, $\mathcal{S}$ does the same reprogramming.

Thus, omitting the events AskH, which probability will be computed later on, the games $\mathbf{G}_4$ and $\mathbf{G}_3$ are indistinguishable.

Game $\mathbf{G}_5$: Simulation of the Server in the Last Step of the Protocol. In this game, we let $\mathcal{S}$ simulate the non-corrupted server in step S4. More precisely, during this game, we consider two cases. If no corruption occurred before $S4$ and all the flows were oracle-generated, the behavior of $\mathcal{S}$ was described in $\mathbf{G}_3$. If, on the other hand, the client has already been corrupted before starting $S4$, or if a flow was non-oracle-generated, the simulation is done as follows.

If the client is either corrupted or impersonated by the adversary who has decrypted Y^* to obtain the Y sent in $Auth$, then the server recovers the password used (by the corruption or by the decryption list) and he verifies the Diffie-Hellman sent by the client. If it is correct, then the simulator asks a GoodPwd query for the server (otherwise, the latter is given an error message). If the password is correct, then the server is given the same key as the client; otherwise, he is given an error message.

If the client is impersonated by the adversary who has sent anything else, we abort the game. This happens only if it has guessed Y by chance, which happens with negligible probability.

Finally, if the server is corrupted during S4, the adversary is given y and Y. More precisely, the simulator recovers the password of the server and gives something consistent with the lists to $\mathcal{A}$. Thus, $\mathbf{G}_5$ and $\mathbf{G}_4$ are indistinguishable.

Game $\mathbf{G}_6$: Simulation of the Server from the Beginning of the Protocol. In this game, we let $\mathcal{S}$ simulate the non-corrupted players from the beginning of the protocol. We have already seen how $\mathcal{S}$ simulates the client. The simulation, for a non-corrupted server, is done as follows.

In S2, the server sends a random Y^* (chosen without asking the encryption oracle). If he gets corrupted, the simulator recovers his password, and can then provide the adversary with adequate y and Y with the help of the encryption and decryption lists. The simulation of S4 has already been described.

$\mathbf{G}_6$ is indistinguishable from $\mathbf{G}_5$, since if the two players remain honest until the end of the game, they have the same key depending on their passwords and nothing else in $\mathbf{G}_3$. And the case in which one of the two gets corrupted has been dealt with in the two former games, and the execution doesn't depend on the value of Y^*, recalling that the encryption is $G \to G$ such that there is always a plaintext corresponding to a ciphertext.

Game $\mathbf{G}_7$: Summary of the Simulation and Replacement of the Hybrid Queries. Here we modify the previous game by replacing the hybrid queries GoodPwd and SamePwd with their ideal versions. If a session aborts or terminates, then $\mathcal{S}$ reports it to $\mathcal{A}$.

Figure 3 sums up the simulation until this point and describes completely the behavior of the simulator. At the beginning of a step of the protocol, the player is assumed to be honest (otherwise we don't have to simulate him or her), and he or she can get corrupted at the end of this step. We assume that U3 (1) has to be executed before both U3 (2) and U3 (3). But the two last can be executed in either order. For simplicity, we assume later on that the order is respected.

We show that $\mathbf{G}_7$ is indistinguishable from the ideal game by first recalling the only difference between $\mathbf{G}_6$ and $\mathbf{G}_7$: the GoodPwd queries are replaced by TestPwd queries to the functionality and the SamePwd by NewKey ones. Say that the players have matching sessions if they share the same $ssid$, have two opposite roles (client and server) and agree on the values of X and Y^*.

	Client	Server	Simulation
U1	honest	honest	random x, $X = g^x$
		adversary	
	gets corrupted	honest	reveal x to $\mathcal{A}$
		adversary	
S2	honest	honest	random Y^*
	adversary		
	honest	gets corrupted	learn pw compute y and Y via decryption query reveal X, y, Y to $\mathcal{A}$
	adversary		
U3 (1)	honest	honest	no decryption query on Y^*
		adversary	
	gets corrupted	honest	learn pw compute y and Y via decryption query reveal x, X, Y to $\mathcal{A}$
		adversary	
U3 (2)	honest	honest	use $\mathcal{H}'_1$ for $Auth$
		adversary	GoodPwd(pw) false, use $\mathcal{H}'_1$
			GoodPwd(pw) correct, use $\mathcal{H}_1$
			if pw unknown, abort
	gets corrupted	honest	learn pw compute y and Y via decryption query reveal x, X, Y to $\mathcal{A}$
		adversary	
U3 (3)	honest	honest	use $\mathcal{H}'_0$ for $Auth$
		adversary	GoodPwd(pw) false, use $\mathcal{H}'_0$
			GoodPwd(pw) correct, use $\mathcal{H}_0$
S4	honest	honest	if SamePwd correct, then same key sk
			if SamePwd incorrect, then error message
	adversary		if pw unknown, then abort
			if pw known, DH false, then error
			if pw known, DH correct, GoodPwd(pw) correct, then same key
			if pw known, DH correct, GoodPwd(pw) false, then error

Fig. 3. Simulation and adaptive corruptions

First, if the two players remain honest until the end of the game, they will obtain a random key, both in $\mathbf{G}_7$ and IWE (the ideal game), as there are no TestPwd queries and the sessions remain fresh.

We need to show that a honest client will receive the same key as a honest server in $\mathbf{G}_7$ if and only if it happens in IWE. We first deal with the case of client and server with matching sessions. If they have the same password in $\mathbf{G}_7$, they will receive the same key: if they are honest, their key is given to them from $\mathbf{G}_3$; if the client is honest with a corrupted server, they will receive their key from $\mathbf{G}_4$; and if the client is corrupted, they will receive it from $\mathbf{G}_5$.

In IWE, the functionality will receive two NewSession queries with the same password. If both players are honest, it will not receive any TestPwd query, so that the key will be the same for both of them. And if one is corrupted and a TestPwd query is done (and correct, since they have the same password), then they will also have the same key, chosen by the adversary.

If they don't have the same password in $\mathbf{G}_7$, the server will always be given an error. In IWE, this is simply the definition of the functionality.

We now deal with the case of client and server with no matching sessions. It is clear that in $\mathbf{G}_7$ the session keys of a client and a server in such a case will be independent because they are not set in any of the games. In IWE, the only way that they receive matching keys is that the functionality receives two NewSession queries with the same passwords, and $\mathcal{S}$ sends NewKey queries for these sessions without having sent any TestPwd queries. But if the two sessions do not have a matching conversation, they must differ in either X, Y^* or $\mathcal{A}uth$. The probability that they share the same pair (X, Y^*) is bounded by q_ε^2/q and thus negligible, q_ε being the number of encryption queries to the oracle.

If the client is corrupted until the end of the game, then in $\mathbf{G}_7$, the server recovers the password and uses it in a TestPwd query to the functionality. If it is incorrect, he is given an error message, and if it is correct, he is given the same key as the client (which was chosen by the simulator). This is exactly the behavior of the functionality in IWE.

If the server gets corrupted, we still have a TestPwd query concerning the client in $\mathbf{G}_7$. If the password is correct, the simulator chooses the key, otherwise it is the adversary. The same thing happens in IWE.

4.3 Simulating Executions via the CDH Problem

As in [2], we compute the probability of event AskH with the help of a reduction to the CDH problem, given one CDH instance (A, B). More precisely, AskH means that there exists one session in which we replaced the random oracles $\mathcal{H}_0$ or $\mathcal{H}_1$ by $\mathcal{H}'_0$ or $\mathcal{H}'_1$ respectively and $\mathcal{A}$ asks the corresponding hash query. We thus choose at random one session, denoted by $ssid$, and we inject the CDH instance in this specific session. With probability $1/q_s$ we have chosen the right session. In this specific session $ssid$, we maintain a list Λ_B, and

- the client sets $X = A$;
- the server still chooses Y^* at random, but the behavior of the decryption is modified on this specific input Y^*, whatever the key is, but only for this session $ssid$: choose a random element $\beta \in \mathbb{Z}_q{}^*$ and compute $Y = Bg^\beta$, and store (β, Y) in the list Λ_B, as well as the usual tuple in Λ_ε. If Y already belongs in this list, one aborts as before.

Note that this only affects the critical session $ssid$ and doesn't change anything else. Contrary to the earlier simulation, we do not know the values of x and φ, but they are not needed since the values of K_U and K_S are no longer required to compute the authenticator and the session key: the event AskH raised for this session (X, Y) means that the adversary has queried the random oracles $\mathcal{H}_0$ or $\mathcal{H}_1$ on $U\|S\|X\|Y\|Z$, where $Z = CDH(X, Y)$. By choosing randomly in the list $\Lambda_\mathcal{H}$, we obtain this Diffie-Hellman triple with probability $1/q_h$, where q_h is the number of hash queries. We can then simply look into the list Λ_B for the values β such that $Y = Bg^\beta$: $CDH(X, Y) = CDH(A, Bg^\beta) = CDH(A, B)A^\beta$.

Note however that in case of corruption, we may need to reveal internal states, with x and φ: If the corruption happens before the end of U3, with the publication of $\mathcal{A}uth$, there is no problem since the random oracles will not be replaced by the private oracles, and then the guess for the session was not correct, which contradicts the assumption of good choice. If the corruption happens after the end of U3, with the publication of $\mathcal{A}uth$, there is no problem either:

- the corruption of the client does not reveal any internal state, since she has completed her execution;
- the corruption of the server leads to a "reprogramming" of the public oracles that immediately raises the event AskH if the query had already been asked. We can thus stop our simulation, and extract the Diffie-Hellman value from the list $\Lambda_{\mathcal{H}}$, without having to wait the end of the whole attack game.

Acknowledgments

This work was supported in part by the European Commission through the IST Program under Contract IST-2002-507932 ECRYPT, and by the French ANR-07-SESU-008-01 PAMPA Project.

References

[1] Michel Abdalla and David Pointcheval. Simple password-based encrypted key exchange protocols. In *CT-RSA 2005*, *LNCS* 3376, pages 191–208. Springer-Verlag, Berlin, Germany, February 2005.

[2] 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.

[3] 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.

[4] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In *EUROCRYPT 2000*, *LNCS* 1807, pages 156–171. Springer-Verlag, Berlin, Germany, May 2000.

[5] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In *EUROCRYPT 2000*, *LNCS* 1807, pages 139–155. Springer-Verlag, Berlin, Germany, 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. Entity authentication and key distribution. In *CRYPTO'93*, *LNCS* 773, pages 232–249. Springer-Verlag, Berlin, Germany, August 1994.

[8] Mihir Bellare and Phillip Rogaway. Provably secure session key distribution — the three party case. In *28th ACM STOC*, pages 57–66. ACM Press, May 1996.

[9] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[10] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In *EUROCRYPT 2005*, *LNCS* 3494, pages 404–421. Springer-Verlag, Berlin, Germany, May 2005.

[11] Ran Canetti and Tal Rabin. Universal composition with joint state. In *CRYPTO 2003*, *LNCS* 2729, pages 265–281. Springer-Verlag, Berlin, Germany, August 2003.

[12] Rosario Gennaro and Yehuda Lindell. A framework for password-based authenticated key exchange. In *EUROCRYPT 2003*, *LNCS* 2656, pages 524–543. Springer-Verlag, Berlin, Germany, May 2003.

[13] Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In *CRYPTO 2006*, *LNCS* 4117, pages 142–159. Springer-Verlag, Berlin, Germany, August 2006.

[14] Dennis Hofheinz and Jörn Müller-Quade. Universally composable commitments using random oracles. In *TCC 2004*, *LNCS* 2951, pages 58–76. Springer-Verlag, Berlin, Germany, February 2004.

[15] Jonathan Katz, Rafail Ostrovsky, and Moti Yung. Efficient password-authenticated key exchange using human-memorable passwords. In *EUROCRYPT 2001*, *LNCS* 2045, pages 475–494. Springer-Verlag, Berlin, Germany, May 2001.

[16] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable block ciphers. In *CRYPTO 2002*, *LNCS* 2442, pages 31–46. Springer-Verlag, Berlin, Germany, August 2002.

[17] Philip D. MacKenzie. The PAK suite: Protocols for password-authenticated key exchange. Contributions to IEEE P1363.2, 2002.

[18] Victor Shoup. On formal models for secure key exchange. Technical Report RZ 3120, IBM, 1999.

A The Random Oracle and the Ideal Cipher

In [10], Canetti *et al.* show that there doesn't exist any protocol that UC-emulates $\mathcal{F}_{pwKE}$ in the plain model (i.e. without additional setup assumptions). Here we show how to securely realize a similar functionality without setup assumption but working in the random oracle and ideal cipher models instead.

Random Oracles. The random oracle functionality was already defined by Hofheinz and Müller-Quade in [14]. We present it again in Figure 4 for completeness. It is clear that the random oracle model UC-emulates this functionality.

The functionality $\mathcal{F}_{RO}$ proceeds as follows, running on security parameter k, with parties $P_1,\ldots,P_n$ and an adversary $\mathcal{S}$:

- $\mathcal{F}_{RO}$ keeps a list L (which is initially empty) of pairs of bitstrings.
- Upon receiving a value (sid, m) (with $m \in \{0,1\}^*$) from some party P_i or from $\mathcal{S}$, do:
  - If there is a pair $(m, \tilde{h})$ for some $\tilde{h} \in \{0,1\}^k$ in the list L, set $h := \tilde{h}$.
  - If there is no such pair, choose uniformly $h \in \{0,1\}^k$ and store the pair $(m,h) \in L$.

  Once h is set, reply to the activating machine (i.e., either P_i or $\mathcal{S}$) with (sid, h).

Fig. 4. Functionality $\mathcal{F}_{RO}$

Ideal Cipher [16]. An ideal cipher is a block cipher that takes a plaintext or a ciphertext as input. We describe the ideal cipher functionality $\mathcal{F}_{IC}$ in Figure 5. Notice that the ideal cipher model UC-emulates this functionality. Note that this functionality characterizes a perfectly random permutation, by ensuring injectivity for each query simulation.

The functionality $\mathcal{F}_{IC}$ takes as input the security parameter k, and interacts with an adversary $\mathcal{S}$ and with a set of (dummy) parties $P_1,\ldots,P_n$ by means of these queries:

- $\mathcal{F}_{IC}$ keeps a (initially empty) list L containing 3−tuples of bitstrings and a number of (initially empty) sets $C_{key,sid}$, $M_{key,sid}$.
- **Upon receiving a query (sid, ENC, key, m) (with $m \in \{0,1\}^k$) from some party P_i or $\mathcal{S}$, do:**
  - If there is a 3−tuple $(key, m, \tilde{c})$ for some $\tilde{c} \in \{0,1\}^k$ in the list L, set $c := \tilde{c}$.
  - If there is no such record, choose uniformly c in $\{0,1\}^k - C_{key,sid}$ which is the set consisting of ciphertexts not already used with *key* and *sid*. Next, it stores the 3−tuple $(key, m, c) \in L$ and sets $C_{key,sid} \leftarrow C_{key,sid} \cup \{c\}$.

  Once c is set, reply to the activating machine with (sid, c).
- **Upon receiving a query (sid, DEC, key, c) (with $c \in \{0,1\}^k$) from some party P_i or $\mathcal{S}$, do:**
  - If there is a 3−tuple $(key, \tilde{m}, c)$ for some $\tilde{m} \in \{0,1\}^k$ in L, set $m := \tilde{m}$.
  - If there is no such record, choose uniformly m in $\{0,1\}^k - M_{key,sid}$ which is the set consisting of plaintexts not already used with *key* and *sid*. Next, it stores the 3−tuple $(key, m, c) \in L$ and sets $M_{key,sid} \leftarrow M_{key,sid} \cup \{m\}$.

  Once m is set, reply to the activating machine with (sid, m).

Fig. 5. Functionality $\mathcal{F}_{IC}$