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, $x \stackrel{R}{\leftarrow} G$ 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 (`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 (NewSession, $ssid$, P_i, P_j, pw, role) from P_i:**
  • Send (NewSession, $ssid$, P_i, P_j, role) to $\mathcal{S}$.
  • If this is the first NewSession query, or if it is the second NewSession query and there is a record $(P_j, P_i, pw', \overline{\text{role}}) \in L$, then record $(P_i, P_j, pw, \text{role})$ in L and mark this record fresh.
– **Upon receiving a query (TestPwd, $ssid$, P_i, pw') from the adversary $\mathcal{S}$:**
  If there exists a record of the form $(P_i, P_j, pw, \text{role}) \in L$ 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 (NewKey, $ssid$, P_i, sk) from $\mathcal{S}$, where $|sk| = k$:**
  If there is a record of the form $(P_i, P_j, pw, \text{role}) \in L$, 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 $(P_i, P_j, pw, \text{client}, \text{completed})$ in L, as well as $(ssid, P_i, pw, sk, \text{client}, \text{status}, \text{ready})$ (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 $(P_i, P_j, pw, \text{client}, \text{completed})$ in L, as well as $(ssid, P_i, pw, sk', \text{client}, \text{status}, \text{ready})$ 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 $(P_i, P_j, pw, \text{server})$ and $(P_j, P_i, pw, \text{client})$, set $s = sk$. Otherwise, if the session is fresh and there exists any recorded element of the form $(ssid, P_j, pw', sk', \text{client}, \text{fresh}, \text{ready})$, set $s = sk'$.
    ∗ If $pw = pw'$, send $(ssid, s)$ to P_i record $(P_i, P_j, pw, \text{server}, \text{completed})$ in L, as well as $(ssid, P_i, pw, s, \text{server}, \text{status})$.
    ∗ If $pw \neq pw'$, send $(ssid, \text{error})$ to P_i, record $(P_i, P_j, pw, \text{server}, \text{completed})$ in L, as well as $(ssid, P_i, pw, \text{server}, \text{error}, \text{status})$.
  • If the session is fresh and there doesn't exist any recorded element of the form $(ssid, P_j, pw', sk', \text{client}, \text{fresh}, \text{ready})$, then do not do anything;
  • If the session is interrupted, then send $(ssid, \text{error})$ to player P_i, and record in L $(P_i, P_j, pw, \text{server}, \text{completed})$ and $(sid, P_i, pw, \text{server}, \text{error}, \text{interrupted})$.

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 $P_1, \ldots, P_n$.

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.*

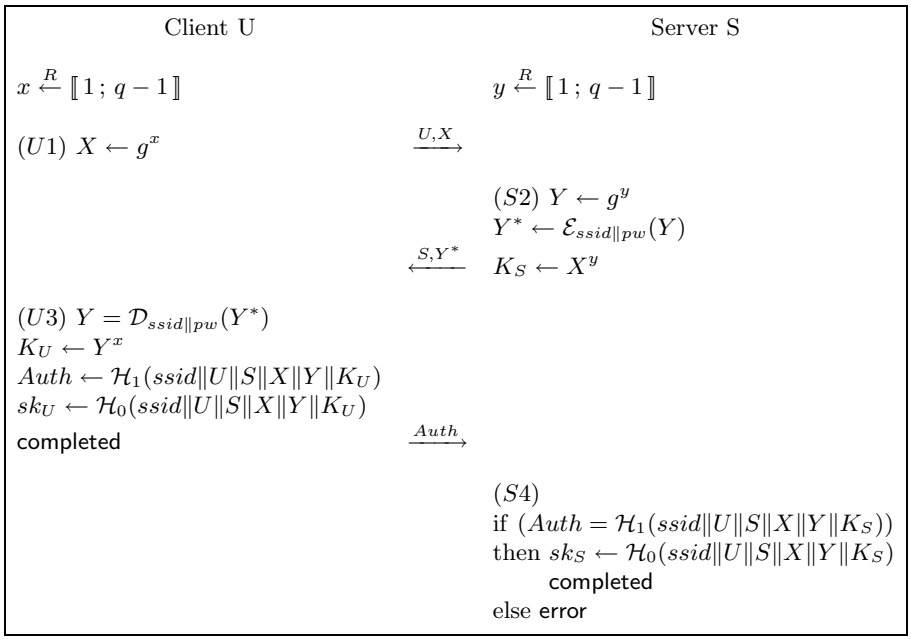

Fig. 2. Client-authenticated two-party password-based key exchange

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 GoodPwd (or a 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 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 $\mathcal{A}uth$ 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	$\mathsf{GoodPwd}(pw)$ false, use $\mathcal{H}'_1$
			$\mathsf{GoodPwd}(pw)$ 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	$\mathsf{GoodPwd}(pw)$ false, use $\mathcal{H}'_0$
			$\mathsf{GoodPwd}(pw)$ correct, use $\mathcal{H}_0$
S4	honest	honest	if $\mathsf{SamePwd}$ correct, then same key sk
			if $\mathsf{SamePwd}$ incorrect, then error message
	adversary		if pw unknown, then abort
			if pw known, DH false, then error
			if pw known, DH correct, $\mathsf{GoodPwd}(pw)$ correct, then same key
			if pw known, DH correct, $\mathsf{GoodPwd}(pw)$ 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 $\mathcal{F}_{IC}$