An extended abstract of this paper appears in ESORICS 2007, J. Biskup and J. Lopez (Eds.), volume 4734 of LNCS, pp. 299-310, Sringer-Verlag, 2007.

Efficient Password-based Authenticated Key Exchange without Public Information *

Jun Shao[1] chn.junshao@gmail.com
Zhenfu Cao[1†] zfcao@cs.sjtu.edu.cn
Licheng Wang[1] wanglc.cn@gmail.com
Rongxing Lu[2] rxlu.cn@gmail.com

[1]Department of Computer Science and Engineering, Shanghai Jiao Tong University
[2]Department of Electrical and Computer Engineering, University of Waterloo

August 19, 2007

Abstract

Since the first password-based authenticated key exchange (PAKE) was proposed, it has enjoyed a considerable amount of interest from the cryptographic research community. To our best knowledge, most of proposed PAKEs based on Diffie-Hellman key exchange need some public information, such as generators of a finite cyclic group. However, in a client-server environment, not all servers use the same public information, which demands clients authenticate those public information before beginning PAKE. It is cumbersome for users. What's worse, it may bring some secure problems with PAKE, such as substitution attack. To remove these problems, in this paper, we present an efficient password-based authenticated key exchange protocol *without* any public information. We also provide a formal security analysis in the non-concurrent setting, including basic security, mutual authentication, and forward secrecy, by using the random oracle model.

1 Introduction

With the rapid-developing of the Internet, people prefer to communicate with each other through the common but insecure channel, rather than traditional methods, such as ordinary mail. It demands a protocol that can provide mutual authentication and generation of a cryptographically-strong (high entropy) shared key for two communicating entities. Password-based authenticated key exchange (PAKE) is a such kind of protocol. In a PAKE, it allows two communicating entities to share a fresh authenticated session key by using a pre-shared human-memorable password. To date, there are two methods to construct a PAKE: the hybrid (i.e., password and public-key) method and the password-only method. In the former method, the two communicating entities share a password and the one additionally knows the public key of the other (see [17, 12]), which demands a secure public-key infrastructure (PKI), thereby arising of issues of user registration, key management, and key revocation. In contrast, in the latter method, the need of a secure PKI can be eliminated, which can make the protocol be more efficient and practical. Note that the pre-shared human-memorable password is low entropy, however, the fresh authenticated session key is high entropy. It seems paradoxical to get a high entropy key by only using a low entropy key. In other words, the latter method seems impossible. However, in 1992, Bellovin and Merritt [3] proposed the first such kind of protocol, named as *Encrypted Key Exchange*, by using a

*Supported by National 863 Project of China, No.2006AA01Z424, National Natural Science Foundation of China, No.60673079 and No.60572155, Research Fund for the Doctoral Program of Higher Education, No.20060248008.

†Corresponding Author.

combination of symmetric and asymmetric cryptographic techniques. In their paper, they proposed two protocols, one is based on RSA [21], and the other is based on ElGamal public key encryption [9].

Due to its simplicity and convenience, password-only authenticated key exchange protocol has received much interest from the cryptographic research community, and most of proposed protocols are based on Bellovin and Merritt's work [4, 20]. However, these protocols have not been proven secure. Until the results in [5, 6], the security proof of PAKE was not treated rigorously. Following these results, a number of provable protocols and improvements have been put forth, in random oracle model [5, 19, 2, 24, 1], in ideal cipher model [6, 2], and in standard model [10, 15, 14, 16, 11]. Most of provable PAKEs based on Diffie-Hellman key exchange need public information [15, 14, 2, 1], such as generators of a finite cyclic group. However, in a client-server environment, not all servers choose the same public information, which would bring some security problems. For example, we use the protocol in [15], which "*do* require that no one know the discrete logarithms of any of the generators with respect to any other" [15]. If an adversary changes the generators (g_1, g_2, h, c, d) to (g_1', g_2', h', c', d'), which he knows the discrete logarithms. As a consequence, a client's password will be revealed after an execution of PAKE with the adversary. And then the adversary can impersonate the client. A natural method to resist this attack (named substitution attack) is to authenticate the public information before beginning PAKE, however, it is cumbersome to clients, and adds complexity to password-only PAKE. The other method is to remove the public information. To our best knowledge, there is no provable PAKE without public information, based on Diffie-Hellman key exchange. In this paper, we propose a such kind of PAKE, which is very efficient (it requires only four and five modular exponentiation computations on client's side and server's side, respectively). Furthermore, we give a formal security analysis in the non-concurrent setting, including basic secure, mutual authentication, and forward secrecy, by using the random oracle model.

1.1 Motivation

In this paper, we focus on the PAKE without public information. But what's the benefit we can get from this kind of PAKE? Firstly, we discuss the disadvantages of the PAKE with public information.

As mentioned above, to resist substitution attack, users must get valid public information. Although users can do it, there still exist some problems, which are described as follows.

- On the one hand, compared with the password, the public information is more complex and larger. For different servers, the public information is different, hence it is a heavy burden for users who store the public information.

- On the other hand, if users will not store the public information, they must get the public information before performing the protocol every time. To our best knowledge, there are two following methods to get the public information.

  - from a public board;
  - from a particular party trusted by communicating parties.

  For the former one, the public board should be maintained by a particular trusted party, whom has to be trusted by all users and all servers, and the data the party maintains will be larger and larger with the number of servers increasing. Furthermore, on the one hand, if the public information for some server changes, which will raise the problems similar to the certificate management problem. On the other hand, if the party is corrupted by some adversary, the adversary can impersonate all users and all servers, such as in the protocols of [15, 14].

  For the latter method, before performing the PAKE with public information, the user must communicate with the particular trusted party, which will increase user's communication and computation

burden. Furthermore, in the PAKE with public information, it requires that the party and server are connectable at the same time. If the user cannot communicate with the party, the PAKE cannot be performed.

Now, we can say that the benefit from the PAKE without public information is to remove the above disadvantages.

1.2 Differences from Previous Work

In fact, the method proposed in this paper is very similar with that in [18, 7], while not the same. On the one hand, in [18], the author proposed a PAKE named Open Key Exchange (OKE), where the server and the client only needs to share the password, while the author focuses on the PAKE based on the RSA problem, not the one based on Diffie-Hellman key exchange. On the other hand, it seems that our proposal belongs to the generic construction in [7], which extends the OKE construction by using *trapdoor hard-to-invert group isomorphisms*. However, in the generic construction, the PAKE needs *six* rounds[1], while our proposal just needs *four* rounds. Furthermore, although the concrete construction based on Diffie-Hellman key exchange in [7] needs the same rounds[2] as our proposal does, the shared information between the client and the server is different from our proposal. The concrete construction requires that the shared information is not only the password, but also the generator of a finite cyclic group, while in our proposal, the shared information is only the password.

In this paper, we aim to propose a provable PAKE based on Diffie-Hellman key exchange, where the client and the server only share the password. Our proposal can be considered as a natural extension of [18, 7].

1.3 Organization

The rest of this paper is organized as follows. In Section 2, we first review the issues related to the security of password-based authenticated key exchange protocol. Then, we propose our protocol and its security analysis in Section 3 and Section 4, respectively. In what follows, we do comparisons our proposal with other PAKEs, and give some discussions on PAKE without public information in Section 5 and Section 6, respectively. Finally, we draw our conclusions in Section 7.

2 Preliminaries

In a password-only authenticated key exchange protocol, there are two entities, say *client* and *server* (denoted by C and S), both holding a secret password drawn from a small password space $\mathcal{P}$. Based on the password, client and server can authenticate each other and generate a fresh session key which is only known to the two of them. There is an active adversary, who controls all communications between client and server, and aims to defeat the goal of the protocol. The adversary can guess a value for the password and use this value in an attempt to impersonate a player, either *on-line* or *off-line*. For the former attack, it can be easily detected by the server after several failed attempts, and the server can halt the account for a while, while the latter one is not the same case due to its independence of the server. Thus, the fundamental security goal of a password-only authenticated key exchange protocol is to be secure against the latter attack. Our formal model of security for password-only authenticated key exchange protocols is based on the "oracle-based" model of Bellare, Pointechaval, and Rogaway [6]. In the following, we recall their definition of their model. For further details, we refer the reader to [6].

[1]We add one round for the client authenticates the server's session key.

[2]We add one round for the client authenticates the server's session key.

2.1 Notes, Initialization

Let I be the set of protocol entities, and C and S be two elements of I, but not fixed. Before running of the protocol, each pair of entities, $C, S \in I$, share a password *pwd*, randomly selected from the password space $\mathcal{P}$. The public information of the protocol, such as the generators of the underlying finite cyclic group, are also specified. However, in our proposal, there does not exist any public information.

2.2 Execution of the Protocol

In a challenge-response protocol, entities' behave in response to received message is determined by the protocol. For each entity, she can execute the protocol multiple times with different entities, which is modeled as an unlimited number of *instances*[3]. We denote the i-th instance of entity C as C^i. Since the adversary is assumed to control all communications among entities, she can interact with entities, which is modeled via access to oracles. The oracle types are as follows:

$Send(C^i, M)$**:** This sends message M to instance C^i. The instance executes and responses as specified by the protocol. This oracle models the active attack.

$Execute(C^i, S^j)$**:** This executes the protocol between instances C^i and S^j honestly and outputs the transcript. This oracle models the passive attack.

$Reveal(I^i)$**:** This outputs the session key sk_I^i of I^i. This oracle models the misuse of session key.

$Test(I^i)$**:** This oracle can be used only once per challenge. The instance I^i generates a random bit b and sends its session key sk_I^i to the adversary if $b = 1$, or a random session key if $b = 0$.

We say that two instances C^i and S^j are *partners* if they both have accepted and hold the same messages sent and received by C^i (or S^j). An instance is said to be *fresh* if the instance has accepted and neither it nor its partner is queried to a `Reveal` oracle.

The notion of semantic security intuitively says that an adversary cannot effectively distinguish between a correct session key and a random session key. This is formally defined via a game, which is described as follows: it initialized by fixing a password *pwd*, randomly chosen from password space $\mathcal{P}$, let the adversary $\mathcal{A}$ ask a polynomial number of queries to the oracles as described above. During the game, the adversary queries a single `Test` oracle on a fresh instance. At the end of game, the adversary $\mathcal{A}$ outputs its guess b' on the bit b selected in the `Test` oracle. We define the advantage of $\mathcal{A}$ to be

$$\mathrm{Adv}_{\mathcal{A}}^{PAKE} = |\Pr|b = b'| - 1/2|.$$

Semantic security means that any efficient adversary's $\mathrm{Adv}_{\mathcal{A}}^{PAKE}$ is no more than $Q(k)/N + \epsilon(k)$, where k is the security parameter, $Q(k)$ is the maximum times of online attacks, N is the size of dictionary, and $\epsilon(\cdot)$ is a negligible function.

2.3 Computational Diffie-Hellman Assumption

Let $\mathbb{G} = \langle g \rangle$ be a finite cyclic group of order a k-bit prime number q. Computational Diffie-Hellman assumption means that there is no probabilistic polynomial time adversary can solve the following problem in $\mathbb{G}$ with non-negligible probability:

On input a tuple (g, g^x, g^y), where $x, y \in Z_q^*$, computing the value g^{xy}.

In the following, we denote ϵ_{cdh} as the probability that the adversary solves the above problem.

[3] The security analysis of our proposal is not in a concurrent setting.

2.4 Decisional Diffie-Hellman Assumption

Let $\mathbb{G} = \langle g \rangle$ be a finite cyclic group of order a k-bit prime number q. Decisional Diffie-Hellman assumption means that there is no probabilistic polynomial time adversary can solve the following problem in $\mathbb{G}$ with non-negligible probability:

On input a quadruple (g, g^x, g^y, g^z), where $x, y, z \in Z_q^*$, outputs the decision whether $g^{xy} = g^z$.

In the following, we denote ϵ_{ddh} as the probability that the adversary solves the above problem.

3 Our Proposal

A high-level description of the protocol is given in Figure 1. Our protocol is in a finite cyclic group $\mathbb{G} = \langle g \rangle$ with a k-bit prime order q, where $\mathbb{G}$ is chosen by client C. $\mathcal{F}_H$ is denoted as the family of universal one-way hash function: $\{0,1\}^* \to \{0,1\}^{k'}$. k and k' are security parameters.

As shown on Figure 1, the protocol runs between a client C and a server S, who initially share a low-entropy secret string pwd, the password, uniformly drawn from the dictionary $\mathcal{P}$, without knowing other public parameters, such as the generator g of the underlying finite cyclic group $\mathbb{G}$, where k and k' are security parameters. Note that all computations are in $\mathbb{G}$.

The protocol consists of the following four flows.

1. The client first chooses a random finite cyclic group $\mathbb{G} = \langle g \rangle$ of order a k-bit prime number q, and selects a random number $r_C \in Z_q^*$, and computes the value $R_C \leftarrow g^{r_C}$, then it sends
$$(\mathbb{G}, q, g, R_C, client)$$
to the server as $Flow_1$.

2. After receiving $Flow_1$, the server first checks whether q is k-bit prime, g and R_C are two members of $\mathbb{G}$ with order q ($g^q \stackrel{?}{=} 1$ and $R_C^q \stackrel{?}{=} 1$). If not, reject $Flow_1$ and abort; otherwise, choose a random number $r_S \in Z_q^*$, and compute
$$R_S \leftarrow g^{r_s},\ R_S^* \leftarrow (R_C)^{pwd} R_S,\ \text{and}\ R' \leftarrow (R_C)^{r_S},$$
then it sends $(R_S^*, server)$ to the client as $Flow_2$.

3. Upon receiving $Flow_2$, the client first checks whether R_S^* is a member of $\mathbb{G}$ with order q ($(R_S^*)^q \stackrel{?}{=} 1$), if not, reject $Flow_2$ and abort; otherwise, choose randomly three hash functions H_0, H_1, H_2 from $\mathcal{F}_H$, and compute
$$R_S' \leftarrow R_S^*(R_C)^{-pwd},\ R \leftarrow (R_S')^{r_C},\ \text{and}\ \alpha \leftarrow H_1(client||server||R_C||R_S'||R),$$
and send (H_0, H_1, H_2, α) to the client as $Flow_3$.

4. On receiving $Flow_3$, the server first checks whether H_0, H_1, H_2 are chosen from $\mathcal{F}_H$, and $\alpha \stackrel{?}{=} H_1(client||server||R_C||R_S||R')$. If not, reject $Flow_3$ and abort; otherwise, compute
$$sk_S \leftarrow H_0(client||server||R_C||R_S||R'),\ \beta \leftarrow H_3(client||server||R_C||R_S||R')$$
which the server sends to the client as $Flow_4$.

5. If $\beta \stackrel{?}{=} H_3(client||server||R_C||R_S'||R)$ holds on the client side, the client computes $sk_C \leftarrow H_0(client||server||R_C||R_S'||R)$, which means that they have successfully exchanged the session key.

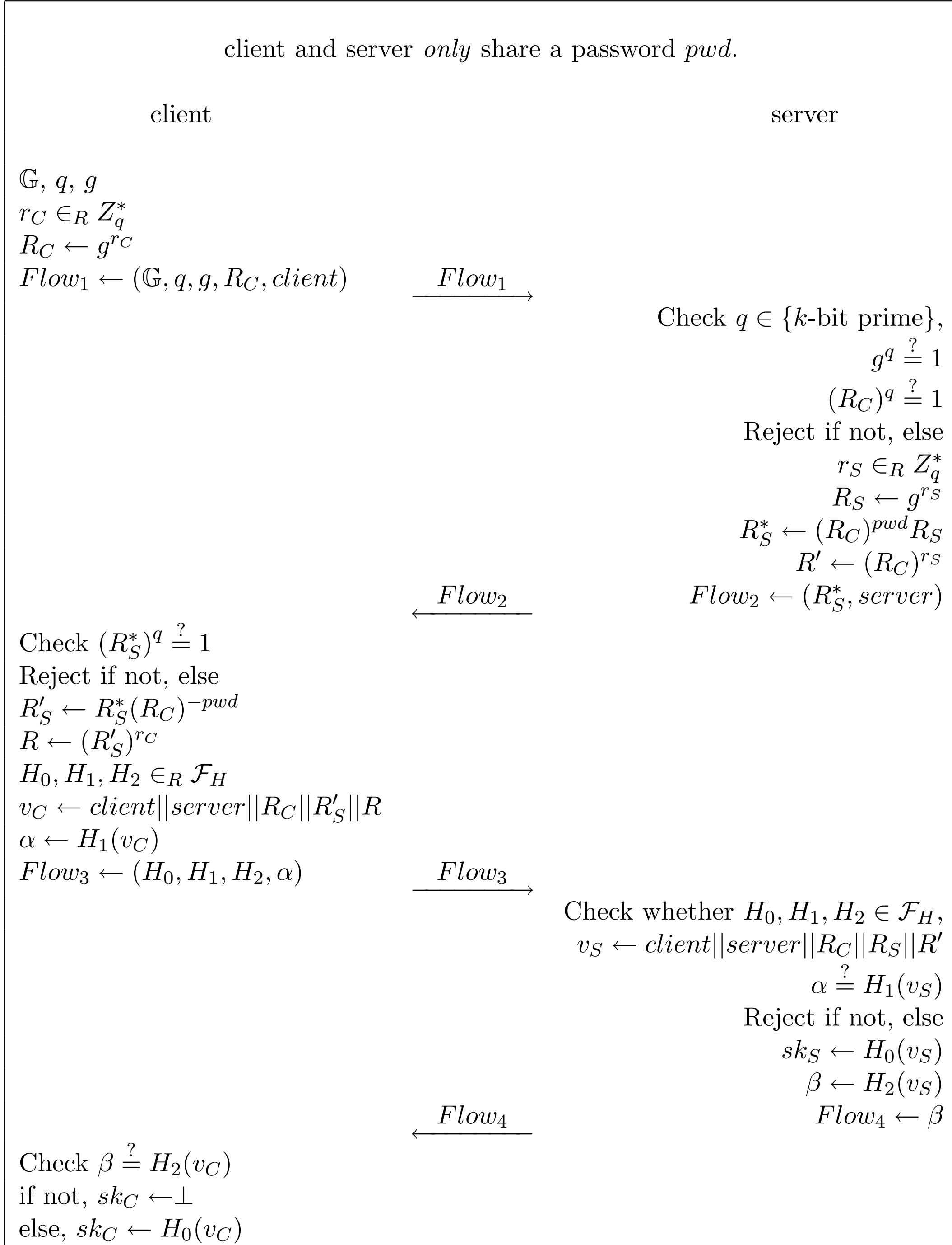

Figure 1: Password-based authenticated key exchange without public information.

3.0.1 Mutual Authentication.

The server authenticates the client by $Flow_3$, and the client authenticates the server by $Flow_4$.

4 Security of Our Protocol

In this section, we deal with the semantic security goal in the non-concurrent setting, including the basic security of the protocol, mutual authentication goal, and forward-secrecy.

4.1 Basic Security

Theorem 1 *Let P be the protocol in Figure 1, where passwords are chosen from a dictionary $\mathcal{P}$ of size N, and let k and k' be the security parameters. Let $\mathcal{A}$ be an adversary which asks q_{ex} queries to `Execute` oracle, q_s queries to `Send` oracle, and q_h queries to the `hash` oracles. Then, in the non-concurrent setting:*

$$Adv_{\mathcal{A}}^{PAKE} < (q_{ex} + q_h + q_s)\epsilon_{ddh} + \frac{q_s}{2^{k'-1}} + \frac{2q_s}{N}$$

Proof: We incrementally define a sequence of experiments starting at the real attack (Experiment Exp_0), and ending up at Experiment Exp_{10}. We define S_i to be the event that $b = b'$ in Experiment Exp_i, where b is the bit involved in the `Test`-query, and b' is the output of the adversary $\mathcal{A}$.

Experiment Exp_0**.** This experiment corresponds to the real attack, which starts by choosing a random password pwd. By definition,

$$\mathrm{Adv}_{P,\mathcal{A}}^{PAKE} = 2\mathrm{Pr}[S_0] - 1. \tag{1}$$

Experiment Exp_1**.** In this experiment, we simulate the hash oracles (H_0, H_1 and H_2, but also three additional hash functions H_3, H_4, and H_5: $\{0,1\}^* \rightarrow \{0,1\}^{k'}$), as usual by maintaining a hash list $\wedge_H$ (and another list $\wedge_{\mathcal{A}}$ containing the hash-queries asked by the adversary itself) (see Figure 2). We also simulate all the instances, as the real players would do, for the `Send`-queries (a list $\wedge_\Psi$ keeps track of the exchanged messages) (see Figure 3), and for the `Execute`, `Reveal` and `Test`-queries (see Figure 4).

From the simulation, we easily see that this experiment is perfectly indistinguishable from the real experiment. Hence,

$$\mathrm{Pr}[S_1] = \mathrm{Pr}[S_0]. \tag{2}$$

— For a hash-query $H_i(x)$ (with $i \in \{0,1,2,3,4,5\}$), such that a record (i,x,y) appears in $\wedge_H$, the answer is y. Otherwise the answer y is defined according to the following rule:

Rule $H^{(1)}$ Choose a random element $y \in \{0,1\}^{k'}$.

The record (i,x,y) is added to $\wedge_H$.

Figure 2: Simulation of the random oracles

Experiment Exp_2**.** In this experiment, queries to the `Execute` oracle are answered as before, except that R is chosen at random from G. The following bounds the effect on the adversary's advantage:

Lemma 1 *The adversary's success probability in Experiment Exp_2 differs by at most ε_{ddh} from its advantage in Experiment Exp_1.*

We answer to the $\texttt{Send}$-queries to the client as follows:

— For a $Send(C^i, \texttt{Start})$-query, the answer $(client, G, g, q, R_C)$ is defined according to the following rule:

Rule $C1^{(1)}$-Choose a random finite cyclic group $G = \langle g \rangle$ of order a k-bit prime number q, and selects a random number θ and compute $R_C \leftarrow g^{\theta}$.

The client instance goes to an expecting state.

— For a $Send(C^i, (server, R_S^*))$-query, if the client instance C^i is in an expecting state, the answer (H_0, H_1, H_2, α) is defined according to the following rule:

Rule $C2^{(1)}$-Choose three random hash functions: H_0, H_1, H_2 from $\mathcal{F}_H$.

Rule $C3^{(1)}$-Compute $R_S^{'} \leftarrow R_S^* {R_C}^{-pwd}$, $R \leftarrow {R_S}^{\theta}$, $\alpha \leftarrow H_1(client||server||R_C||R_S^{'}||R)$

Then the client instance goes to an expecting state.

— For a $Send(C^i, \beta)$-query, if the client instance C^i is in an expecting state, this query is processed according to the following rules:

Rule $C4^{(1)}$-Compute $\beta' \leftarrow H_2(client||server||R_C||R_S^{'}||R)$ and check whether $\beta = \beta'$. If the equality does not hold, the client instance terminates without accepting.

If equality holds, the client instance accepts and goes on, applying the following rule:

Rule $C5^{(1)}$-Compute the session key $sk_C \leftarrow H_0(client||server||R_C||R_S^{'}||R)$.

Finally, the client instance terminates. We also adds $(Flow_1, Flow_2, Flow_3)$ to $\wedge_{\Psi}$.

We answer to the $\texttt{Send}$-queries to the server as follows:

— For a $Send(S^j, (client, G, g, q, R_C))$-query, it is processed according to the following rule:

Rule $S1^{(1)}$-Check whether q is k-bit prime, g is a generator of G with order q. If the equality does not hold, the server instance S^j terminates without accepting.

If equality holds, the server instance S^j accepts and goes on, applying the following rule:

Rule $S2^{(1)}$-Choose a random exponent $\varphi \in Z_q^*$, compute $R_S \leftarrow g^{\varphi}$, $R_S^* \leftarrow {R_C}^{pwd} R_S$, and $R' \leftarrow R_C^{r_S}$

Finally the query is answered with $(server, R_S^*)$, and the server instance goes to an expecting state.

— For a $Send(S^j, (client, H_0, H_1, H_2, \alpha))$-query, it is processed according to the following rule:

Rule $S3^{(1)}$-Compute $\alpha' \leftarrow H_1(client||server||R_C||R_S||R')$, and check whether $\alpha = \alpha'$ and H_0, H_1, H_2 are chosen from $\mathcal{F}_H$. If the equality does not hold, the server instance S^j terminates without accepting.

If equality holds, the server instance S^j accepts and goes on, applying the following rule:

Rule $S4^{(1)}$-Compute $\beta \leftarrow H_2(client||server||R_C||R_S||R')$ and $sk_S \leftarrow H_0(client||server||R_C||R_S||R')$.

Finally the query is answered with $(server, \beta)$, and the server instance terminates. We also adds $(Flow_1, Flow_2, Flow_3, Flow_4)$ to $\wedge_{\Psi}$.

Figure 3: Simulation of the $\texttt{Send}$-queries.

—For a $Execute(C^i, S^j)$-query, it proceed as follows:

Choose a random finite cyclic group $G = \langle g \rangle$ with a k-bit prime q, and three random hash functions H_0, H_1, H_2 from $\mathcal{F}_H$.

$\theta \stackrel{R}{\leftarrow} Z_q^*$; $R_C \leftarrow g^\theta$;

$\varphi \stackrel{R}{\leftarrow} Z_q^*$; $R_S \leftarrow g^\varphi$; $R_S^* \leftarrow {R_C}^{pwd} R_S$;

$R \leftarrow {R_S}^\theta$;

$\alpha \leftarrow H_1(client||server||R_C||R_S||R)$;

$\beta \leftarrow H_2(client||server||R_C||R_S||R)$;

$sk_C \leftarrow H_0(client||server||R_C||R_S||R)$; $sk_S \leftarrow sk_C$;

The query is answered with $((G, g, q, R_C, client), (R_S^*, server), (H_0, H_1, H_2, \alpha), \beta)$.

—For a $Reveal(I^i)$-query, it is answered the session key computed by the instance I^i, if the latter has accepted. Otherwise $\perp$.

—For a $Test(I^i)$-query, it is processed as follows:

Get the session key from $Reveal(I^i)$, and flips a coin b. If $b = 1$, the output is the value of the session key, otherwise, the output is a random value drawn from $\{0, 1\}^{k'}$.

Figure 4: Simulation of the `Execute`, `Reveal` and `Test`-queries.

—For a $Execute(C^i, S^j)$-query, it proceed as follows:

Choose three random hash functions H_0, H_1, H_2 from $\mathcal{F}_H$.

$\theta \stackrel{R}{\leftarrow} Z_q^*$; $R_C \leftarrow u^\theta$;

$\varphi \stackrel{R}{\leftarrow} Z_q^*$; $R_S \leftarrow v^\varphi$; $R_S^* \leftarrow {R_C}^{pwd} R_S$;

$R \leftarrow w^{\theta\varphi}$;

$\alpha \leftarrow H_1(client||server||R_C||R_S||R)$;

$\beta \leftarrow H_2(client||server||R_C||R_S||R)$;

$sk_C \leftarrow H_0(client||server||R_C||R_S||R)$; $sk_S \leftarrow sk_C$;

The query is answered with $((G, g, q, R_C, client), (R_S^*, server), (H_0, H_1, H_2, \alpha), \beta)$.

Figure 5: Simulation of the `Execute`-queries.

Proof: The simulator uses the adversary as a black box to distinguish Diffie-Hellman tuples from random tuples. Given tuple (g, u, v, w) and group $\mathbb{G} = \langle g \rangle$, the query to `Execute` oracle is answered as in Figure 5.

By a random self-reducibility property of the Diffie-Hellman problem, if (g, u, v, w) is a Diffie-Hellman tuple, this is an exact simulation of Experiment Exp_1; on the other hand, if it is a random tuple, this is an exact simulation of Experiment Exp_2. □

Since the adversary can issue q_{ex} `Execute` queries at most, hence,

$$|\Pr[S_2] - \Pr[S_1]| \leq q_{ex}\epsilon_{ddh}. \tag{3}$$

Experiment Exp_3**.** In this experiment, we replace the random oracle H_i $(i \in \{0, 1, 2\})$ for computing α, β, sk_C and sk_S for all sessions generated via an `Execute` oracle query. More precisely, we use private random oracle H_{i+3}, and in the `Execute`-queries, one gets $\alpha \leftarrow H_4(client||server||R_C||R_S^*)$, $\beta \leftarrow H_5(client||server||R_C||R_S^*)$, and $sk_C, sk_S \leftarrow H_3(client||server||R_C||R_S^*)$.

Since the resulting values are random to $\mathcal{A}$ as in those the Experiment Exp_2, it is clear that the Experiment Exp_3 is perfectly indistinguishable from the Experiment Exp_2. Hence,

$$\Pr[S_3] = \Pr[S_2]. \tag{4}$$

Experiment Exp_4**.** In this experiment, we ensure that all accepted α will come from either the simulator, or an adversary that has correctly guessed the value of $R_S^{'}$. We reach this aim by modifying the following rule:

Rule $S2^{(4)}$-Check whether $\alpha = \alpha'$, where $\alpha' = H_1(client||server||R_C||R_S||R')$. If the equality does hold, check if $(1, client||server||R_C||R_S||R', \alpha) \in \wedge_{\mathcal{A}}$ or $((client, R_C), (server, R_S^*), \alpha) \in \wedge_{\Psi}$. If these two later tests fail, then reject α: terminate, without accepting. If this rule does not make the server to terminate, the server accepts and moves on.

The two experiments Exp_4 and Exp_3 are perfectly indistinguishable unless the server rejects a valid α. In experiment Exp_4, we ensure that all accepted α will come from either the simulator, or an adversary that has correctly guessed the value of $R_S^{'}$. However, in experiment Exp_3, the accepted α may come from other method, i.e., the adversary has correctly guessed the value of α directly. Hence,

$$|\Pr[S_4] - \Pr[S_3]| \leq \frac{q_s}{2^{k'}}. \tag{5}$$

Experiment Exp_5**.** In this experiment, we ensure that all accepted β will come from either the simulator, or an adversary that has correctly guessed the value of R_S. We reach this aim by modifying the following rule:

Rule $C2^{(5)}$-Check whether $\beta = \beta'$, where $\beta' = H_2(client||server||R_C||R_S^{'}||R)$. If the equation does hold, check if $(2, client||server||R_C||R_S^{'}||R, \beta) \in \wedge_{\mathcal{A}}$ or $((client, R_C), (server, R_S^*), \alpha, \beta) \in \wedge_{\Psi}$. If these two later tests fail, then reject β: terminate, without accepting. If this rule does not make the client to terminate, the client accepts and moves on.

The two experiments Exp_5 and Exp_4 are perfectly indistinguishable unless the server rejects a valid β. In experiment Exp_5, we ensure that all accepted β will come from either the simulator, or an adversary that has correctly guessed the value of R_S'. While in experiment Exp_4, the accepted β may come from other two method, i.e., the adversary has correctly guessed the value of β directly, or gets the value of R_S' by using offline dictionary attack[4]. Hence,

$$|\Pr[S_5] - \Pr[S_4]| \leq q_s(\frac{1}{2^{k'}} + \epsilon_{cdh}) < q_s(\frac{1}{2^{k'}} + \epsilon_{ddh}). \tag{6}$$

[4]If the adversary can solve the CDH problem, with the value of α, he can launch the offline dictionary attack to get the value of *pwd*.

Experiment Exp_6**.** In this experiment, we modifying the $Send(S^j, (client, G, g, q, R_C))$-query by maintaining a list $\wedge_{\mathcal{E}}$. We use the following rule:

Rule $S2^{(6)}$-Choose a random exponent $\varphi \in Z_q^*$, compute $R_S \leftarrow g^{\varphi}$ and $R_S^* \leftarrow R_C^{pwd} R_S$. If such a tuple $(g, q, \varphi, pwd, R_C, R_S^*)$ does not exist in the list $\wedge_{\mathcal{E}}$, add it to the list $\wedge_{\mathcal{E}}$.

It is clear that Experiments Exp_6 and Exp_5 are perfectly indistinguishable. Hence,

$$\Pr[S_6] = \Pr[S_5]. \tag{7}$$

Experiment Exp_7**.** In this experiment, we modifying the $Send(C^j, (server, R_S^*))$-query by maintaining a list $\wedge_{\mathcal{D}}$. We use the following rule:

Rule $C2^{(7)}$-Choose three random hash functions: H_0, H_1, H_2 from $\mathcal{F}_H$, and compute $R_S^{'} \leftarrow R_S^* {R_C}^{-pwd}$, $R \leftarrow {R_S}^{\theta}$, $\alpha \leftarrow H_1(client||server||R_C||R_S^{'}||R)$. If such a tuple $(g, q, \theta, pwd, R_C, R_S^*)$ does not exist in the list $\wedge_{\mathcal{D}}$, add it to the list $\wedge_{\mathcal{D}}$.

It is clear that Experiments Exp_7 and Exp_6 are perfectly indistinguishable. Hence,

$$\Pr[S_7] = \Pr[S_6]. \tag{8}$$

Experiment Exp_8**.** In this experiment, we abort the game wherein the adversary may have been lucky in guessing the password pwd, then used it to get $R_S^{'}$, and asked the query to the oracle H_1. We use the following rule:

Rule $S2^{(8)}$-Check whether $\alpha = \alpha'$, where $\alpha' = H_1(client||server||R_C||R_S||R')$. If the equality does hold, check if $(1, client||server||R_C||R_S||R', \alpha) \in \wedge_{\mathcal{A}}$ or $((client, R_C), (server, R_S^*), \alpha) \in \wedge_{\Psi}$. It the equality does hold, check if $(g, p, *, pwd, R_C, R_S^*) \in \wedge_{\mathcal{D}}$. If the record is not found, abort the game. Otherwise, the server accepts and moves on.

This rule ensures that all accepted α come from the simulator. The two experiments Exp_8 and Exp_7 are perfectly indistinguishable unless the server rejects a valid α, which implies that the adversary has been lucky in guessing the password pwd. Hence,

$$|\Pr[S_8] - \Pr[S_7]| \leq \frac{q_s}{N}. \tag{9}$$

Experiment Exp_9**.** In this experiment, we abort the game wherein the adversary may have been lucky in guessing the password pwd, and used it to get R_S and R', and asked the query to the oracle H_2. We use the following rule:

Rule $C2^{(9)}$-Check whether $\beta = \beta'$, where $\beta' = H_2(client||server||R_C||R_S^{'}||R)$. If the equality does hold, check if $(2, client||server||R_C||R_S^{'}||R, \beta) \in \wedge_{\mathcal{A}}$ or $((client, R_C), (server, R_S^*), \alpha, \beta) \in \wedge_{\Psi}$. It the equality does hold, check if $(g, p, *, pwd, R_C, R_S^*) \in \wedge_{\mathcal{E}}$. If the record is not found, abort the game. Otherwise, the server accepts and moves on.

This rule ensures that all accepted β come from the simulator. The two experiments Exp_9 and Exp_8 are perfectly indistinguishable unless the server rejects a valid β, which implies that the adversary has been lucky in guessing the password pwd. Hence,

$$|\Pr[S_9] - \Pr[S_8]| \leq \frac{q_s}{N}. \tag{10}$$

Experiment Exp_{10}**.** In this experiment, we do not compute the values of α, β, and the session key sk using the oracles H_0, H_1, H_2, but using the private oracles H_3, H_4 and H_5. We use the following rules:

Rule $C3^{(10)}$-$\alpha \leftarrow H_4(client||server||R_C||R_S^*)$.

Rule $C5^{(10)}$-$sk_C \leftarrow H_3(client||server||R_C||R_S^*)$.

Rule $S4^{(10)}$-$\beta \leftarrow H_5(client||server||R_C||R_S^*)$, and $sk_S \leftarrow H_3(client||server||R_C||R_S^*)$.

In this experiment, the session keys are random, independent from any other data (from an information theoretical point of view, since H_3, H_4 and H_5 are private random oracles). Then

$$\Pr[S_{10}] = 1/2 \tag{11}$$

The experiments Exp_{10} and Exp_9 are indistinguishable unless the following event `AskH` occurs: $\mathcal{A}$ queries the hash functions H_i ($i \in \{0,1,2\}$) on $client||server||R_C||R_S^{'}||R$ or on $client||server||\ R_C||R_S||R'$, that is on the common value $client||server||R_C||R_S||\mathrm{CDH}(R_C, R_S)$. Since the probability of this event is $\Pr[AskH] \leq q_h\epsilon_{cdh} < q_h\epsilon_{ddh}$. Hence,

$$|\Pr[S_{10}] - \Pr[S_9]| < q_h\epsilon_{ddh}. \tag{12}$$

Combining (1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (12), and (11), we obtain,

$$\mathrm{Adv}_{P,\mathcal{A}}^{PAKE} < (q_{ex} + q_h + q_s)\epsilon_{ddh} + \frac{q_s}{2^{k'-1}} + \frac{2q_s}{N}$$

as required. ■

4.2 Mutual Authentication

The following theorem shows that our protocol ensures mutual authentication, that is, a server/client instance will never accept a non-corresponding/non-expected client/server instance with non-negligible probability. We denote that `AuthC`/`AuthS` is the probability that a server/client instance accepts a non-corresponding/non-expected client/server instance.

Theorem 2 *Let us consider our protocol, where $\mathcal{P}$ is a finite dictionary of size N equipped with the uniform distribution. Let $\mathcal{A}$ be an an adversary against the security of our protocol, with less than q_s `Send` queries, q_{ex} `Execution` queries, and q_h `hash` queries. Then in the non-concurrent setting, we have*

$$\texttt{AuthC} < (q_{ex} + q_s)\epsilon_{ddh} + \frac{q_s}{2^{k'-1}} + \frac{q_s}{N},$$

$$\texttt{AuthS} < (q_{ex} + q_s)\epsilon_{ddh} + \frac{q_s}{2^{k'-1}} + \frac{2q_s}{N}.$$

Proof. We use the proof presented in above theorem, and denote that $\mathrm{AuthC}_i/\mathrm{AuthS}_i$ is the probability that a server/client instance accepts a non-corresponding/non-expected client/server instance in Experiment Exp_i.

For `AuthC`, since

$$\texttt{AuthC} = \mathrm{AuthC}_0,$$

and in Experiment Exp_8, $\mathrm{AuthC}_8 = 0$, and Equations (2), (3), (4), (5), (6), (7), (8), and (9) extends to

$$\begin{array}{ll}
\mathrm{AuthC}_1 = \mathrm{AuthC}_0, & |\mathrm{AuthC}_2 - \mathrm{AuthC}_1| \leq q_{ex}\epsilon_{ddh}, \\
\mathrm{AuthC}_3 = \mathrm{AuthC}_2, & |\mathrm{AuthC}_4 - \mathrm{AuthC}_3| \leq \frac{q_s}{2^{k'}}, \\
|\mathrm{AuthC}_5 - \mathrm{AuthC}_4| < q_s(\frac{1}{2^{k'}} + \epsilon_{ddh}), & \mathrm{AuthC}_6 = \mathrm{AuthC}_5, \\
\mathrm{AuthC}_7 = \mathrm{AuthC}_6, & |\mathrm{AuthC}_8 - \mathrm{AuthC}_7| \leq \frac{q_s}{N}.
\end{array}$$

Then we have

$$\texttt{AuthC} \leq (q_{ex} + q_s)\epsilon_{ddh} + \frac{q_s}{2^{k'-1}} + \frac{q_s}{N}$$

For `AuthS`, since

$$\texttt{AuthS} = \mathrm{AuthS}_0,$$

and in Experiment Exp_9, $\text{AuthC}_9 = 0$, and Equations (2), (3), (4), (5), (6), (7), (8), (9), and (10) extends to

$$\begin{array}{ll} \text{AuthS}_1 = \text{AuthC}_0, & |\text{AuthS}_2 - \text{AuthS}_1| \leq q_{ex}\epsilon_{ddh}, \\ \text{AuthS}_3 = \text{AuthC}_2, & |\text{AuthS}_4 - \text{AuthS}_3| \leq \frac{q_s}{2^{k'}}, \\ |\text{AuthS}_5 - \text{AuthC}_4| < q_s(\frac{1}{2^{k'}} + \epsilon_{ddh}), & \text{AuthS}_6 = \text{AuthS}_5, \\ \text{AuthS}_7 = \text{AuthC}_6, & |\text{AuthS}_8 - \text{AuthS}_7| \leq \frac{q_s}{N}, \\ |\text{AuthS}_9 - \text{AuthS}_8| \leq \frac{q_s}{N}. & \end{array}$$

Then we have

$$\text{AuthS} \leq (q_{ex} + q_s)\epsilon_{ddh} + \frac{q_s}{2^{k'-1}} + \frac{2q_s}{N}$$

■

4.3 Forward Secrecy

In this section, in order to deal with forward secrecy, we introduce a new kind of query named the `Corrupt`-query [2]:

Corrupt(I): This query models the adversary $\mathcal{A}$ have succeeded at getting the password *pwd* of the entity I. However, $\mathcal{A}$ does not get internal data of I.

Now, we say an instance is a `fresh` instance if before the `Corrupt`-query has been asked, the instance has accepted and neither it nor its partner is queried to a `Reveal` Oracle.

Forward-secrecy ensures that the adversary can not get any information about the session keys established before the password *pwd* is revealed. We use the same game in Section 2 to define forward-secrecy, and denote the advantage of $\mathcal{A}$ to be

$$Adv_{\mathcal{A}}^{PAKE-FS} = |Pr|b = b'| - 1/2|.$$

Forward-secrecy means that any efficient adversary's $Adv_{\mathcal{A}}^{PAKE-FS}$ is negligible.

Theorem 3 *Let us consider our protocol, where $\mathcal{P}$ is a finite dictionary of size N equipped with the uniform distribution. Let $\mathcal{A}$ be an an adversary against the security of our protocol, with less than q_s `Send` queries, q_{ex} `Execution` queries, and q_h `hash` queries. Then in the non-concurrent setting, we have*

$$Adv_{\mathcal{A}}^{PAKE-FS} < (q_{ex} + q_h + q_s)\epsilon_{ddh} + \frac{q_s}{2^{k'-1}} + \frac{2q_s}{N}.$$

We use the proof on a similar sequence of experiments as before, but just modifying some rules.

Rule $S2^{(8)}$- Check whether $\alpha = \alpha'$, where $\alpha' = H_1(client||server||R_C||R_S||R')$. If the equality does hold, check if $(1, client||server||R_C||R_S||R', \alpha) \in \wedge_{\mathcal{A}}$ or $((client, R_C), (server, R_S^*), \alpha) \in \wedge_{\Psi}$. It the equality does hold, check if $(g, p, *, pwd, R_C, R_S^*) \in \wedge_{\mathcal{D}}$. If the record is not found and `Corrupted=false`, abort the game. Otherwise, the server accepts and moves on.

Rule $C2^{(9)}$- Check whether $\beta = \beta'$, where $\beta' = H_2(client||server||R_C||R_S^{'}||R)$. If the equality does hold, check if $(2, client||server||R_C||R_S^{'}||R, \beta) \in \wedge_{\mathcal{A}}$ or $((client, R_C), (server, R_S^*), \alpha, \beta) \in \wedge_{\Psi}$. It the equality does hold, check if $(g, p, *, pwd, R_C, R_S^*) \in \wedge_{\mathcal{E}}$. If the record is not found and `Corrupted=false`, abort the game. Otherwise, the server accepts and moves on.

Rule $C3^{(10)}$- If `Corrupted=false`, $\alpha \leftarrow H_4(client||server||R_C||R_S^*)$;

otherwise, $\alpha \leftarrow H_4(client||server||R_C||R_S^{'}||R)$.

Rule $C5^{(10)}$- If `Corrupted=false`, $sk_C \leftarrow H_3(client||server||R_C||R_S^*)$;

otherwise, $sk_C \leftarrow H_3(client||server||R_C||R_S'||R)$.

Rule $S4^{(10)}$- If `Corrupted=false`, $\beta \leftarrow H_5(client||server||R_C||R_S^*)$,

and $sk_S \leftarrow H_3(client||server||R_C||R_S^*)$;

otherwise, $\beta \leftarrow H_5(client||server||R_C||R_S'||R')$,

and $sk_S \leftarrow H_3(client||server||R_C||R_S'||R')$.

As a consequence, we can get the similar result:

$$\mathrm{Adv}_{\mathcal{A}}^{PAKE-FS} < (q_{ex} + q_h + q_s)\epsilon_{ddh} + \frac{q_s}{2^{k'-1}} + \frac{2q_s}{N}.$$

■

5 Comparison

In this section, we will compare our proposal with the scheme in [13] (named IEEE) and the scheme in [1] (named AP05). From our viewpoint, the hash functions are not the public information, but the common sense, like the operator "+" in algebra. Since in our proposal, no matter which special finite cyclic group $\mathbb{G} = \langle g \rangle$ is, we can always use the hash function $SHA-1$ only. For example, set $H_0 : SHA-1(client||server||R_C||R_S||R||0)$, $H_1 : SHA-1(client||server||R_C||R_S||R||1)$, and $H_2 : SHA-1(client||server||R_C||R_S||R||2)$.

Table 1: Comparison of PAKEs between with and without public information.

		Our proposal	IEEE	AP05
public information		None	$\mathbb{G}$, g, q, $(\mathcal{E}_k, \mathcal{D}_k)$	$\mathbb{G}$, g, q, M, N
the total number of round		4	3	2
Authentication		Mutual	Unilateral	None
Computation Costs	Client's side	$4T_e$ [a] $+ 1T_m$ [b]	$2T_e$	$3T_e + 2T_m$
	Server's side	$5T_e + 1T_m$	$2T_e$	$3T_e + 2T_m$
Communication Costs [c]	Client's side	6	2	1
	Server's side	3	2	1

[a] Time for a modular exponentiation computation

[b] Time for a modular multiplication computation

[c] Since the schemes all work in a finite cyclic group $\mathbb{G} = \langle g \rangle$ of order a k-bit prime number q, hence, we just consider the total number of data unit.

From Table 1, compared with IEEE and AP05, our proposal is a little bit inefficient than these two protocols.

- Its computational overhead is five more modular exponentiation computations than that in IEEE, and three more modular exponentiation computations than that in AP05. Since in our proposal, the server has to check the validity of $Flow_1$, and the client has to check the validity of $Flow_2$, but IEEE and AP05 do not need to do this.
- Its communication costs on client's side are more than that in IEEE and AP05. Since in our proposal, the client's terminal does need transmit the parameters. If we want to reduce the length of transmitting data, we can use the cyclic finite group on the elliptic curve.

- Since our proposal provides full functions including mutual authentication, while IEEE and AP05 do not. Hence, the total number of round in our proposal is more than that in IEEE and AP05.

6 Discussion

6.0.1 The Parameters Can Be Reused.

Now, let us think more about our new kind of PAKE. We can find that there is no need for the client's terminal to generate new parameters each time. Since every server can perform the same as the proposed scheme suggests, hence, it allows the client to choose its own parameters once and re-use them for several different servers. In fact, if the client has a device with the parameters, then the same parameters can be used every time. We think it is very flexible and pretty attractive to users.

6.0.2 Generating And Testing The Parameters.

In our proposal, the client's terminal should generate $\mathbb{G}, q, g$, and the server's terminal should verify these parameters. For the client's terminal, since the user can reuse the parameters, the time for generating the parameters is not a problem in our proposal. For the server's terminal, checking whether an element g in a cyclic finite group is a generator with a prime order q is fast, which just needs a exponentiation computation in the underlying cyclic finite group ($g^q \stackrel{?}{=} 1$). On the other hand, there exist fast algorithms to test primality [23, 22]. As a result, the time for testing the parameters is not a problem in our proposal, neither.

6.0.3 Is There Existing PAKE Without Public Information?

The answer is "Yes". Most PAKEs based on RSA [3, 19, 24] can be considered as the PAKE without public information, since the public key of RSA (n, e) is chosen by the client, and the client sends them to the server. However, our proposal is the first provable-secure PAKE without public information, only sharing password, based on Diffie-Hellman key exchange.

6.0.4 Can All PAKEs Be Changed Into The PAKE Without Public Information?

The answer is also "Yes". If the protocol just needs one generator of the underlying finite cyclic group, it can be changed into the PAKE without public information by the method in our proposal. If the protocol needs more than one generators, it should need more communication and computation to compute the generators, such as performing a standard Diffie-Hellman key exchange [8] to get a generator.

7 Conclusion

In this paper, to remove the disadvantages raised by getting valid public information, we have proposed an efficient password-based authenticated exchange protocol without public information. Furthermore, we gave its security proof in the non-concurrent setting, including basic security, mutual authentication, and forward secrecy, by using the random oracle model.

Compared with the PAKEs with public information, our proposal is a little bit inefficient in terms of computational complexity. However, since the parameters can be reused in our proposal, it is very flexible and attractive to users.

References

[1] M. Abdalla and D. Pointcheval. Simple Password-based Encrypted Key Exchange Protocols. In *CT-RSA 2005*, LNCS 3376, pp. 191-208, 2005. 1, 5

[2] E. Bresson, O. Chevassut, and D. Pointcheval. Security Proofs for an Efficient Password-Based Key Exchange. In *Proc. of the 10th ACM Conference on Computer and Communication Security*, pp. 241-250, 2003. 1, 4.3

[3] S.M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proc. of the IEEE Symposium on Research in Secruity and Privacy*, pp. 72-84, 1992. 1, 6.0.3

[4] S.M. Bellovin and M. Merritt. Augmented encrypted key exchange: A passowrd-based protocol secure against dictionary attacks and password file compromise. In *ACM CCS 1993*, pp. 244-250, 1993. 1

[5] V. Boyko, P. MacKenzie, and S. Patel. Provably secure password authenticated key exchange using Diffie-Hellman. In *EUROCRYPT 2000*, LNCS 1807, pp. 156-171, 2000. 1

[6] M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attack. In *EUROCRYPT 2000,* LNCS 1807, pp. 139-155, 2000. 1, 2

[7] D. Catalano, D. Pointcheval, and T. Pornin. IPAKE: Isomorphisms for Password-based Authenticated Key Exchange. In *CRYPTO 2004*, LNCS 3152, pp. 477-493, 2004. 1.2

[8] W. Diffie and M. Hellman. New Directions in Cryptography. IEEE Trans. Info. Theory, vol. 22, no. 6, 1976, pp. 644-654. 6.0.4

[9] T. ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, vol. IT-31, no.4, 1985, pp. 469-472. 1

[10] O. Goldreich and Y. Lindell. Session-key generation using human passwords only. In *CRYPTO 2001*, LNCS 2139, pp. 408-432, 2001. 1

[11] R. Gennaro and Y. Lindell. A framework for password-based authenticated key exchange. In *EUROCRYPT 2003*, LNCS 2656, pp. 524-542, 2003. 1

[12] S. Halevi and H. Krawczyk. Public-Key Cryptography and Password Protocols. *ACM Trans. on Info. and Sys. Security*, vol. 2, no. 3, 1999, pp. 230-268. 1

[13] IEEE Standard 1363-2000. Standard Specifications for Public Key Cryptography. IEEE. Available from http://grouper.ieee.org/groups/1363, August 2000. 5

[14] K. Kobara and H. Imai. Pretty-simple password-authenticated key-exchange under standard assumptions. *IEICE Trans.*, vol. E85-A, no. 10, 2002, pp. 2229-2237. 1, 1.1

[15] J. Katz, R. Ostrovsky, and M. Yung. Efficient Password-Authenticated Key Exchange Using Human-Memorable Passwords. In *EUROCRYPT 2001*, LNCS 2045, pp. 475-494, 2001. 1, 1.1

[16] J. Katz, R. Ostrovsky, and M. Yung. Forward Screcy in Password-only Key Exchange Protocols. In *SCN 2002*, LNCS 2576, pp. 29-44, 2003. 1

[17] T. M. A. Lomas, L. Gong, J. H. Saltzer, and R. M. Needham. Reducing Risks from Poorly-Chosen Keys. *ACM Operating Systems Review*, vol. 23, no. 5, 1989, pp. 14-18. 1

[18] S. Lucks. Open Key Exchange: How to Defeat Dictionary Attacks Without Encrypting Public Keys. In *Proc. of thhe Security Protocols Workshop*, LNCS 1361, pp. 79-90, 1997. 1.2

[19] P. MacKenzie, S. Patel, and R. Swaminathan. Password-authenticated key exchange based on RSA. In *ASIACRYPT 2000*, LNCS 1976, pp. 599-613, 2000. 1, 6.0.3

[20] D.P. Jablon. Strong password-only authenticated key exchange, *SIGCOMM Computer Communications Review*, vol. 26, no. 5, pp. 5-26, 1996. 1

[21] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*, vol. 21, no. 2, 1978, pp. 120-126. 1

[22] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal of Computing*, vol. 6, no. 1, 1977. 6.0.2

[23] E. W. Weisstein. Primality Testing Is Easy. http://mathworld.wolfram.com/news/2002-08-07/primetest/ 6.0.2

[24] M. Zhang. New Approaches to Password Authenticated Key Exchange Based on RSA. In *ASIACRYPT 2004*, LNCS 3329, 2004, pp. 230-244. 1, 6.0.3