Forward Secrecy in Password-Only Key Exchange Protocols

JONATHAN KATZ[1,4] RAFAIL OSTROVSKY[2] MOTI YUNG[3]

[1] Department of Computer Science, University of Maryland (College Park)
jkatz@cs.umd.edu
[2] Telcordia Technologies, Inc.
rafail@research.telcordia.com
[3] Department of Computer Science, Columbia University
moti@cs.columbia.edu
[4] Work done while at Columbia University

Abstract. Password-only authenticated key exchange (PAKE) protocols are designed to be secure even when users choose short, easily-guessed passwords. Security requires, in particular, that the protocol cannot be broken by an off-line dictionary attack in which an adversary enumerates all possible passwords in an attempt to determine the correct one based on previously-viewed transcripts. Recently, provably-secure protocols for PAKE were given in the idealized random oracle/ideal cipher models [2, 8, 19] and in the standard model based on general assumptions [11] or the DDH assumption [14].

The latter protocol (the *KOY protocol*) is currently the only known efficient solution based on standard assumptions. However, only a proof of basic security for this protocol has appeared. In the basic setting the adversary is assumed not to corrupt clients (thereby learning their passwords) or servers (thereby modifying the value of stored passwords). Simplifying and unifying previous work, we present a natural definition of security which incorporates the more challenging requirement of *forward secrecy*. We then demonstrate via an explicit attack that the KOY protocol as originally presented is *not* secure under this definition. This provides the first natural example showing that forward secrecy is a strictly stronger requirement for PAKE protocols. Finally, we present a slight modification to the KOY protocol which prevents the attack and — as the main technical contribution of this paper — rigorously prove that the modified protocol achieves forward secrecy.

1 Introduction

Protocols allowing mutual authentication of two parties and generation of a cryptographically-strong shared key (*authenticated key exchange*) underly most secure interactions on the Internet. Indeed, it is near-impossible to achieve any level of security over an unauthenticated network without mutual authentication and key-exchange protocols. The former are necessary because one needs to know "with whom one is communicating", while the latter are required because cryptographic techniques (such as encryption, etc.) are useless without a shared key which must be periodically refreshed. Furthermore, high-level protocols are frequently developed and analyzed using the assumption of "authenticated channels"; this assumption cannot be realized without a secure mechanism for implementing such channels using previously-shared information.

Client-server authentication requires *some* information to be shared between client and server or else there is nothing distinguishing the client from other parties in the network. The classical example of shared information is a high entropy, cryptographic *key*; this key can then be used, e.g., for message authentication or digital signatures. Indeed, the first systematic and formal treatments of authenticated key exchange [10, 6, 3, 4, 1, 20] assumed that participants shared cryptographically-strong information: either a secret key [6, 3, 4, 1, 20] or public keys [10, 1, 20]. Under these strong setup assumptions, many protocols for the two-party case have been designed and proven secure [6, 3, 1, 20].

If the client is ultimately a human user, however, it is unrealistic to expect the client to store (i.e., remember) a high entropy key. Instead, it is more reasonable to assume that the client stores only a low entropy *password*. Protocols which are secure when users share keys are often demonstrably insecure when users share passwords. For example, a challenge-response protocol in which the client sends r and the server replies with $x = f_K(r)$ (where K is the shared key) is secure only when the entropy of K is sufficiently large. When K has low entropy, an eavesdropper who monitors a single conversation $\langle r, x \rangle$ can determine (with high probability) the value of K, off-line, by trying all possibilities until a value K' is found such that $f_{K'}(r) = x$.

When a password is shared, we distinguish between the *hybrid* (i.e., password/public-key) model [15, 12, 7], in which the client and server share a password and the client additionally knows the public key of the server, and the *password-only* model [5, 2, 8] in which the client and server share only a password (although public information may be available to all parties in the network[1]). In both cases, it is important to design protocols which are secure against off-line dictionary attacks in which an adversary enumerates all possible passwords, one-by-one, in an attempt to determine the correct password based on previously-recorded transcripts. Consideration of such attacks is crucial if security is to be guaranteed even when users of the protocol choose passwords poorly.

Password-only protocols (even when public information is assumed) have many practical advantages over protocols designed for the hybrid model. The password-only model eliminates the need for a PKI, thereby avoiding issues of user registration, key management, and key revocation. Furthermore, eliminating a PKI means that an on-line, trusted certification authority (CA) is no longer needed; note that access to the CA is often a performance bottleneck as the number of users becomes large. In the password-only model, once public

[1] The existence of such information is a typical (though not universal) assumption. For example, in Diffie-Hellman key exchange [9] the group $\mathbb{G}$ and a generator of $\mathbb{G}$ are often assumed to be public.

information is established new users may join at any time and do not need to inform anyone else of their presence. Finally, in the password-only model there is no "secret key" associated with the public parameters. This eliminates the risk that compromise of a single participant will compromise the security of the entire system.

Motivation for our work. Formal definitions of security for password-only authenticated key exchange (PAKE) are non-trivial, and are the subject of ongoing research. At least three different frameworks for this setting have been proposed [8, 2, 11]. Further complicating the issue is that, in all three frameworks, two levels of security can be distinguished depending on whether forward secrecy is required (we discuss the definition of forward secrecy in Section 2.1). Making matters worse, the various definitions of forward secrecy are subtly different from one another and hence a multitude of security notions exists; as an example, in [2] alone four distinct notions of forward secrecy are mentioned! One of our goals is to present a concise definition which simplifies and unifies previous work.

It is also important to design PAKE protocols which provably achieve forward secrecy. A number of PAKE protocols achieving basic security are known. The first such protocol was given by Bellovin and Merritt [5], although they give only heuristic arguments for its security. More recently, the first formal proofs of basic security for PAKE protocols have been given in the random oracle [8, 19] or ideal cipher models [2]. Subsequently, PAKE protocols were designed and proven secure in the standard model: Goldreich and Lindell [11] construct a protocol based on general assumptions which does not require public parameters, and Katz, Ostrovsky, and Yung [14] give a protocol (the *KOY protocol*) based on the decisional Diffie-Hellman assumption. Subsequently, other protocols with provable security in the random oracle model have been proposed [16, 17].

Only some of the above protocols are known to achieve forward secrecy.[2] Forward secrecy is claimed for the protocol of [2], although no proof is given. A full proof of forward secrecy for the protocol of [8] has subsequently appeared [18]. In the standard model, Goldreich and Lindell [11] prove forward secrecy of their protocol. We point out, however, that the definitions of forward secrecy considered in these previous works are weaker than the definition presented here (see discussion in Section 2.1).

Our contributions As mentioned above, a number of definitions have been proposed for forward secrecy of PAKE protocols. However, we feel that none of these definitions adequately capture all realistic attacks. Building on the framework of Bellare, et al. [2] and extending the definition of forward secrecy contained therein, we propose a new definition of forward secrecy in the weak corruption model. We believe our definition better captures the underlying issues than previous definitions; in fact, we show concrete examples of potentially damaging attacks which are not prevented under previous definitions of forward secrecy but which are handled by our definition.

[2] Here and throughout the rest of the paper, we consider only the weak corruption model [2]. Our terminology is explained in greater detail in Section 2.1.

In Section 3.1, we demonstrate via an explicit attack that the KOY protocol as presented in [14] does *not* achieve forward secrecy with respect to our definition. The attack represents a potentially serious threat to the protocol in practice. Of additional interest, our attack shows a natural separation between the notions of basic and forward secrecy in the PAKE setting. We suggest a modification of the KOY protocol which prevents this attack, and (as our main technical contribution) give a complete proof of forward secrecy for this modified protocol in Section 4.

2 Definitions

Due to space limitations, we assume the reader is familiar with the "oracle-based" model of Bellare, Pointcheval, and Rogaway [2] (building on [3, 4]) as well as their definition of basic security. Our point of departure from their definition is with regard to forward secrecy, so we only summarize those aspects of their model necessary for understanding the present work. For further details, we refer the reader to [13].

Participants, passwords, and initialization. We assume for simplicity a fixed set of protocol participants (also called principals or users) each of which is either a client $C \in \mathsf{Client}$ or a server $S \in \mathsf{Server}$, where Client and Server are disjoint. Each $C \in \mathsf{Client}$ has a password pw_C. Each $S \in \mathsf{Server}$ has a vector $PW_S = \langle pw_{S,C} \rangle_{C \in \mathsf{Client}}$ which contains the passwords of each of the clients. We assume that all clients share passwords with all servers.

Before the protocol is run, an initialization phase occurs during which global, public parameters are established and passwords pw_C are chosen for each client. We assume that passwords for each client are chosen independently and uniformly[3] at random from the set $\{1, \ldots, N\}$, where N is a constant which is fixed independently of the security parameter. At the outset of the protocol, the correct passwords are stored at each server so that $pw_{S,C} = pw_C$ for all $C \in \mathsf{Client}$ and $S \in \mathsf{Server}$.

Execution of the protocol. In the real world, a protocol determines how principals behave in response to signals from their environment. In the model, these signals are sent by the adversary. Each principal can execute the protocol multiple times with different partners; this is modeled by allowing each principal an unlimited number of *instances* with which to execute the protocol. We denote instance i of user U as Π_U^i. A given instance may be used only once. Each instance Π_U^i has associated with it the variables state_U^i, term_U^i, acc_U^i, used_U^i, sid_U^i, pid_U^i, and sk_U^i; the function of these variables is as in [2].

The adversary is assumed to have complete control over all communication in the network. The adversary's interaction with the principals (more specifically, with the various instances) is modeled via access to *oracles* which are described in detail below. Local state (i.e., values of state, term, etc.) is maintained for each

[3] Our analysis extends to handle arbitrary distributions, including users with interdependent passwords.

instance with which the adversary interacts; this state is not directly visible to the adversary. The state of an instance may be updated during an oracle call, and the oracle's output will typically depend upon this state. The oracle types are:

- $\mathsf{Send}(U, i, M)$ — This sends message M to instance Π_U^i. The oracle outputs the reply generated by this instance.
- $\mathsf{Execute}(C, i, S, j)$ — This executes the protocol between instances Π_C^i and Π_S^j (where $C \in \mathsf{Client}$ and $S \in \mathsf{Server}$) and outputs the transcript of this execution. This represents occasions when the adversary passively eavesdrops on a protocol execution.
- $\mathsf{Reveal}(U, i)$ — This outputs the current value of session key sk_U^i.
- $\mathsf{Corrupt}(\cdots)$ — We discuss this oracle in Section 2.1. This oracle is not present in the definition of basic security, and is used to define forward secrecy.
- $\mathsf{Test}(U, i)$ — This query is allowed only once, at any time during the adversary's execution. A random bit b is generated; if $b = 1$ the adversary is given sk_U^i, and if $b = 0$ the adversary is given a random session key.

Correctness. As in [2].

Partnering. We say that two instances Π_U^i and $\Pi_{U'}^j$ are *partnered* if: (1) $U \in \mathsf{Client}$ and $U' \in \mathsf{Server}$, or $U \in \mathsf{Server}$ and $U' \in \mathsf{Client}$; (2) $\mathsf{sid}_U^i = \mathsf{sid}_{U'}^j \neq \textsc{null}$; (3) $\mathsf{pid}_U^i = U'$ and $\mathsf{pid}_{U'}^j = U$; and (4) $\mathsf{sk}_U^i = \mathsf{sk}_{U'}^j$. (This slightly clarifies the notion of partnering in [2].)

2.1 Forward Secrecy

As mentioned earlier, our main departure from [2] is in our definition of forward secrecy. To completely define this notion, two orthogonal components must be specified: the nature of the Corrupt oracle and what it means for an adversary to succeed in breaking the protocol.

The corruption model. The Corrupt oracle models corruption of participants by the adversary. Since the adversary in our setting already has the ability to impersonate (i.e., send messages on behalf of) parties in the network, "corruption" in our setting involves learning or modifying secret information stored by a participant. Three specific possibilities are: (1) Corruption of player U may allow the adversary to learn the internal state of all (active) instances of U. (2) Corruption of client C may allow the adversary to learn pw_C. Finally, (3) corruption of server S may allow the adversary to modify passwords stored on S; that is, to change password $pw_{S,C}$ (for some C) to any desired value.

As defined in [2], the *strong corruption model* allows attacks (1)–(3) while the *weak corruption model* allows attacks (2) and (3) only. However, this terminology is not universally accepted. In particular, the weak corruption models of [8, 11] allow only attack (2); i.e., they do not allow the adversary to install bogus passwords on servers.

Here, we focus on the weak corruption model and allow the adversary to both learn passwords of clients and to change passwords stored on servers (in other words, we allow attacks (2) and (3)). However, our precise formalization of the Corrupt oracle differs from that of [2] in that we consider each such attack separately. Formally, oracle $\mathsf{Corrupt}_1$ returns pw_C when given $C \in \mathsf{Client}$ as input. Oracle call $\mathsf{Corrupt}_2(S, C, pw)$ (for $S \in \mathsf{Server}$ and $C \in \mathsf{Client}$) sets $pw_{S,C} := pw$. We emphasize that the adversary can install different passwords on different servers for the same client. In the definition of [2], an adversary who installs a password $pw_{S,C}$ also learns the "actual" password pw_C; we make no such assumption here. Note that in the case of a poorly-administered server, it may be easy to modify users' passwords without learning their "actual" passwords.

We choose to focus on the weak corruption model rather than the strong corruption model for two reasons. First, fully satisfactory definitions of security in the latter model have not yet appeared; in particular, we know of no protocols satisfying any reasonable definition of forward secrecy in this model. Part of the problem is that current definitions in the strong corruption model are overly restrictive; in fact, a generalization of the argument given in [2] shows that forward secrecy (under their definition) is impossible in the strong corruption case. We therefore leave this as a subject for future research. Secondly, weak corruption is more relevant in practice: it will often be easier to compromise a machine *after completion of* protocol execution than to compromise a machine *during* protocol execution. Installing bogus passwords on servers is also a realistic threat if, e.g., a server's password file is encrypted but not otherwise protected from malicious tampering.

Advantage of the adversary. Our most significant departure from previous work is with regard to the definition of the adversary's success. Previous definitions have a number of flaws which allow attacks on supposedly "secure" protocols; our definition aims at correcting these flaws.

In any definition of forward secrecy, the adversary succeeds if it can guess the bit b used by the Test oracle when this oracle is queried on a "fresh" instance. Differences among the definitions arise due to different definitions of "freshness". Note that *some* definition of freshness is necessary for any reasonable definition of security; if no such notion were defined the adversary could always succeed by, for example, submitting a Test query for an instance for which it had already submitted a Reveal query.

In the definition of [2], an instance Π_U^i is *fresh*[4] unless one of the following is true: (1) at some point, the adversary queried $\mathsf{Reveal}(U, i)$; (2) at some point, the adversary queried $\mathsf{Reveal}(U', j)$ where $\Pi_{U'}^j$ and Π_U^i are partnered; or (3) the adversary made a Corrupt query before the Test query and at some point the adversary queried $\mathsf{Send}(U, i, M)$ for some M. This definition has been adopted for all subsequent work of which we are aware.

This definition, however, considers "secure" a protocol in which revealing one client's password enables the adversary to impersonate a *different* client. This is

[4] In [2] this is called "fs-fresh" but since we focus on forward secrecy we use the abbreviated name.

so (under the above definition) because every instance with which the adversary interacts following a $\mathsf{Corrupt}$ query is unfresh and no guarantees are given with regard to unfresh instances. As another example of a flaw in the definition, consider an adversary who installs password $pw_{S,C}$ for client C at server S, interacts with server S, and *is then able to impersonate* C *to server* $S' \neq S$. Under the above definition, such a protocol could be considered secure! In fact, an attack of this sort represents a real threat: in Section 3.1, we demonstrate precisely this type of attack against the original KOY protocol.

We are more careful in our definition of freshness. Under our definition, an instance Π_U^i with $\mathsf{pid}_U^i = U'$ is fresh unless one of the following is true:

- The adversary queried $\mathsf{Reveal}(U,i)$ or $\mathsf{Reveal}(U',j)$ where $\Pi_{U'}^j$ and Π_U^i are partnered.
- The adversary queried $\mathsf{Corrupt}_1(U)$ or $\mathsf{Corrupt}_1(U')$ before the Test query and at some point the adversary queried $\mathsf{Send}(U,i,M)$ for some M.
- The adversary queried $\mathsf{Corrupt}_2(U,U',pw)$ before the Test query and at some point the adversary queried $\mathsf{Send}(U,i,M)$ for some M.

Note that, in contrast with previous definitions (as explained above), exposing the password of user U no longer automatically results in instances of user $U' \neq U$ being unfresh.

We have not yet defined what we mean by a secure protocol. Note that a PPT adversary can always succeed by trying all passwords one-by-one in an on-line impersonation attack. Informally, we say a protocol is secure if this is the best an adversary can do. Formally, an instance Π_U^i represents an *on-line attack* if both the following are true: (1) at some point, the adversary queried $\mathsf{Send}(U,i,M)$ and (2) at some point, the adversary queried $\mathsf{Reveal}(U,i)$ or $\mathsf{Test}(U,i)$. In particular, instances with which the adversary interacts via $\mathsf{Execute}$ calls are not counted as on-line attacks. The number of on-line attacks represents a bound on the number of passwords the adversary could have tested in an on-line fashion. This motivates the following definition:

Definition 1. *Protocol* P *is a* secure PAKE protocol achieving forward secrecy *if, for all* N *and for all* PPT *adversaries* $\mathcal{A}$ *making at most* $Q(k)$ *on-line attacks, there exists a negligible function* $\varepsilon(\cdot)$ *such that* $\mathsf{Adv}_{\mathcal{A},P}(k) \leq Q(k)/N + \varepsilon(k)$.

In particular, the adversary can (essentially) do no better than guess a single password during each on-line attempt. Calls to the $\mathsf{Execute}$ oracle, which are not included in $Q(k)$, are of no help to the adversary in breaking the security of the protocol; this means that passive eavesdropping and off-line dictionary attacks are of (essentially) no use.

Some previous definitions of security consider protocols secure as long as the adversary can do no better than guess a *constant* number of passwords in each on-line attempt. We believe the strengthening given by Definition 1 (in which the adversary can guess only a *single* password per on-line attempt) is an important one. The space of possible passwords is small to begin with, so any degradation in security should be avoided if possible. This is not to say that protocols which do not meet this definition of security should never be used; however, before using such a protocol, one should be aware of the constant implicit in the proof of security.

An examination of the security proofs for some protocols [2, 8, 19] shows that these protocols achieve the stronger level of security given by Definition 1. However, security proofs for other protocols [11, 17] are inconclusive, and leave open the possibility that more than one password can be guessed by the adversary per on-line attack. In at least one case [21], an explicit attack is known which allows an adversary to guess two passwords per on-line attack.

3 The KOY Protocol

Due to space limitations, we include only those details of the KOY protocol that are necessary for understanding our attack, our modification, and our proof of security. We refer the reader to [14, 13] for more details.

A high-level description of the protocol is given in Figure 1. During the initialization phase, public information is established as follows: for a given security parameter k, primes p, q are chosen such that $|q| = k$ and $p = 2q+1$; these values define a group $\mathbb{G}$ in which the decisional Diffie-Hellman (DDH) assumption is believed to hold. Values g_1, g_2, h, c, d are selected at random from $\mathbb{G}$, and a function $H : \{0,1\}^* \to \mathbb{Z}_q$ is chosen at random from a universal one-way hash family. The public information consists of (a description of) $\mathbb{G}$, the values g_1, g_2, h, c, d, and the hash function H.

As part of the initialization phase, passwords are chosen randomly for each client and stored at each server. We assume that all passwords lie in (or can be mapped in a one-to-one fashion to) $\mathbb{G}$. As an example, if passwords lie in the range $\{1, \ldots, N\}$, password pw can be mapped to $g_1^{pw} \in \mathbb{G}$; this will be a one-to-one mapping for reasonable values of N.

When a client $\mathit{Client} \in \mathsf{Client}$ wants to connect to a server $\mathit{Server} \in \mathsf{Server}$, the client begins by running a key-generation algorithm for a one-time signature scheme, giving VK and SK. The client chooses random $r_1 \in \mathbb{Z}_q$ and computes $A = g_1^{r_1}$, $B = g_2^{r_1}$, and $C = h^{r_1} \cdot pw_C$. The client then computes $\alpha = H(\mathit{Client}|\mathsf{VK}|A|B|C)$ and sets $D = (cd^{\alpha})^{r_1}$. These values are sent to the server as the first message of the protocol.

Upon receiving the first message $\langle \mathit{Client}|\mathsf{VK}|A|B|C|D \rangle$, the server first chooses random $x_2, y_2, z_2, w_2 \in \mathbb{Z}_q$, computes $\alpha' = H(\mathit{Client}|\mathsf{VK}|A|B|C)$, and sets $E = g_1^{x_2} g_2^{y_2} h^{z_2} (cd^{\alpha'})^{w_2}$. Additionally, a random $r_2 \in \mathbb{Z}_q$ is chosen and the server computes $F = g_1^{r_2}$, $G = g_2^{r_2}$, and $I = h^{r_2} \cdot pw_{S,C}$ (where $pw_{S,C}$ is the password stored at the server for the client named in the incoming message). The server then computes $\beta = H(\mathit{Server}|E|F|G|I)$ and sets $J = (cd^{\beta})^{r_2}$. These values are sent to the client as the second message of the protocol.

Upon receiving the second message $\langle \mathit{Server}|E|F|G|I|J \rangle$, the client chooses random $x_1, y_1, z_1, w_1 \in \mathbb{Z}_q$, computes $\beta' = H(\mathit{Server}|E|F|G|I)$, and sets $K = g_1^{x_1} g_2^{y_1} h^{z_1} (cd^{\beta'})^{w_1}$. The client then signs $\beta'|K$ using SK. The value K and the resulting signature are sent as the final message of the protocol. At this point,

	Public: $\mathbb{G}, g_1, g_2, h, c, d \in \mathbb{G};\ H : \{0,1\}^* \to \mathbb{Z}_q$	
Client		Server
$(\mathsf{VK}, \mathsf{SK}) \leftarrow \mathcal{K}(1^k)$		
$r_1 \leftarrow \mathbb{Z}_q$		
$A := g_1^{r_1};\ B := g_2^{r_1}$		
$C := h^{r_1} \cdot pw_C$		
$\alpha := H(\mathit{Client}\,\vert \mathsf{VK} \vert A \vert B \vert C)$		
$D := (cd^{\alpha})^{r_1}$	$\xrightarrow{\mathit{Client} \mid \mathsf{VK} \mid A \mid B \mid C \mid D}$	
		$x_2, y_2, z_2, w_2, r_2 \leftarrow \mathbb{Z}_q$
		$\alpha' := H(\mathit{Client}\,\vert \mathsf{VK} \vert A \vert B \vert C)$
		$E := g_1^{x_2} g_2^{y_2} h^{z_2} (cd^{\alpha'})^{w_2}$
		$F := g_1^{r_2};\ G := g_2^{r_2}$
		$I := h^{r_2} \cdot pw_{S,C}$
		$\beta := H(\mathit{Server}\,\vert E \vert F \vert G \vert I)$
	$\xleftarrow{\mathit{Server} \mid E \mid F \mid G \mid I \mid J}$	$J := (cd^{\beta})^{r_2}$
$x_1, y_1, z_1, w_1 \leftarrow \mathbb{Z}_q$		
$\beta' := H(\mathit{Server}\,\vert E \vert F \vert G \vert I)$		
$K := g_1^{x_1} g_2^{y_1} h^{z_1} (cd^{\beta'})^{w_1}$		
$\mathsf{Sig} \leftarrow \mathsf{Sign}_{\mathsf{SK}}(\beta' \mid K)$	$\xrightarrow{K \mid \mathsf{Sig}}$	if $\mathsf{Vrfy}_{\mathsf{VK}}(\beta \mid K, \mathsf{Sig}) = 1$
		$C' := C / pw_{S,C}$
$I' := I / pw_C$		$\mathsf{sk}_S := A^{x_2} B^{y_2} (C')^{z_2} D^{w_2} K^{r_2}$
$\mathsf{sk}_C := E^{r_1} F^{x_1} G^{y_1} (I')^{z_1} J^{w_1}$		else $\mathsf{sk}_S :=$ NULL

Fig. 1. The KOY protocol.

the client accepts and computes the session key by first computing $I' = I/pw_C$ and then setting $\mathsf{sk}_C = E^{r_1} F^{x_1} G^{y_1} (I')^{z_1} J^{w_1}$.

Upon receiving the final message $\langle K \vert \mathsf{Sig} \rangle$, the server checks that Sig is a valid signature of $\beta \vert K$ under VK (where β is the value previously used by the server). If so, the server accepts and computes the session key by first computing $C' = C/pw_{S,C}$ and then setting $\mathsf{sk}_S = A^{x_2} B^{y_2} (C')^{z_2} D^{w_2} K^{r_2}$. If the received signature is not valid, the server does not accept and the session key remains NULL.

Although omitted in the above description, we assume that users always check that incoming messages are well-formed; e.g., when the server receives the first message it verifies that $\mathit{Client} \in \mathsf{Client}$ and that $A, B, C, D \in \mathbb{G}$. If an ill-formed message is received, the receiving party terminates without accepting and the session key remains NULL.

3.1 An Attack on the KOY Protocol

Here, we demonstrate that the KOY protocol is not forward secure under our definition by showing an explicit attack. Our results do not impact the basic security of the protocol.

Fix a client *Client*. The attack begins by having the adversary impersonate *Client* to a server S using an arbitrary password, say $pw = g_1$. Thus, the adversary chooses r_1 and VK, constructs a message $\langle \mathit{Client} | \mathsf{VK} | A | B | C | D \rangle$ (using password g_1), and sends this message to server S. The server will respond with a message $\langle S | E | F | G | I | J \rangle$ (which is computed using password $pw_{S,Client} = pw_{Client}$). Next, the adversary changes the password stored at S for *Client* (i.e., $pw_{S,Client}$) to the value g_1 using oracle call $\mathsf{Corrupt}_2(S, \mathit{Client}, g_1)$. Finally, the adversary computes the final message of the protocol by choosing x_1, y_1, z_1, w_1 and sending $\langle K | \mathsf{Sig} \rangle$. If the adversary later obtains sk_S (via a Reveal query), we claim the adversary can determine pw_{Client} and hence successfully impersonate *Client* to a different server $S' \neq S$.

To see this, note that when S computes the session key after receiving the final message of the protocol it uses $pw_{S,Client} = g_1$. Thus,

$$\mathsf{sk}_S = A^{x_2} B^{y_2} (C/g_1)^{z_2} D^{w_2} K^{r_2} = g_1^{r_1 x_2} g_2^{r_1 y_2} h^{r_1 z_2} (cd^{\alpha})^{r_1 w_2} K^{r_2} = E^{r_1} K^{r_2}.$$

Since the adversary can compute E^{r_1}, the adversary can use sk_S to determine the value K^{r_2}. Now, using exhaustive search through the password dictionary, the adversary can identify pw_{Client} as that unique value for which:

$$F^{x_1} G^{y_1} (I/pw_{Client})^{z_1} J^{w_1} = K^{r_2}.$$

3.2 Modifying the Protocol to Achieve Forward Security

The reason the attack described in the previous section succeeds is precisely because the password used by the server changes in the middle of protocol execution. To ensure that this does not happen, we modify the protocol so that the password $pw_{S,Client}$ used to construct the server's response is *saved as part of the state information* for that instance. When the server later computes the session key (for a given instance), it uses the value of $pw_{S,Client}$ stored as part of the state information (for that particular instance) rather than using the "actual" value of $pw_{S,Client}$ stored in "long term memory" (e.g., the password file). Note that if multiple server-instances are active concurrently, it is possible for different passwords to be in use for each instance. Although this modification is a simple one, it is a crucial change that must be made in any implementation of the protocol.

The proof of forward secrecy for the modified protocol is complex. In fact, we view the main technical contribution of this paper as a detailed, full proof of forward secrecy; this is the first such proof (without random oracles/ideal ciphers) of forward secrecy in the model of [2].

4 Proof of Forward Secrecy

Although the modification given in the previous section seems to thwart known attacks (in particular the one given in Section 3.1) the rigorous proof of forward secrecy is challenging. We present such a proof here; note that this is the first such proof for *any* PAKE protocol under the (natural) definition of forward secrecy appearing here.

Theorem 1. *Under the DDH assumption, the modified KOY protocol is a secure PAKE protocol achieving forward secrecy.*

Proof. Due to space limitations, only a sketch of the proof is given here. A complete proof appears in the full version of this work [13].

As in [14], the adversary has access to oracles Send_i $(0 \leq i \leq 3)$ representing the four different types of messages (including an "initiate" message) that can be sent. We imagine a simulator that runs the protocol for adversary A. When the adversary completes its execution and outputs b', the simulator can tell whether A succeeds by checking whether (1) a single Test query was made on an instance Π_U^i; (2) instance Π_U^i is fresh; and (3) $b' = b$. Success of the adversary is denoted by event fsSucc. We refer to the real execution of the experiment as P_0.

In experiment P_0', the adversary does *not* succeed if any of the following occur:

1. A verification key VK used by the simulator in responding to a Send_0 query is repeated.
2. The adversary forges a new, valid message/signature pair for any verification key used by the simulator in responding to a Send_0 query.
3. A value β used by the simulator in responding to Send_1 queries is repeated.
4. A value β used by the simulator in responding to a Send_1 query (with $\mathit{msg\text{-}out} = \langle \mathit{Server}|E|F|G|I|J\rangle$) is equal to a value β used by the simulator in responding to a Send_2 query (with $\mathit{msg\text{-}in} = \langle \mathit{Server}'|E'|F'|G'|I'|J'\rangle$) and furthermore it is the case that $\langle \mathit{Server}|E|F|G|I\rangle \neq \langle \mathit{Server}'|E'|F'|G'|I'\rangle$.

Since the probability that any of these events occur is negligible (assuming the security of the signature scheme and the universal one-way hash function), we have $\mathsf{fsAdv}_{A,P_0}(k) \leq \mathsf{fsAdv}_{A,P_0'}(k) + \varepsilon(k)$ for some negligible function $\varepsilon(\cdot)$.

In experiment P_1, upon receiving oracle query $\mathsf{Execute}(\mathit{Client}, i, \mathit{Server}, j)$, values C and I are chosen independently at random from $\mathbb{G}$. The simulator then checks whether $pw_{\mathit{Client}} = pw_{\mathit{Server},\mathit{Client}}$. If so, the session keys are computed as

$$\mathsf{sk}^i_{\mathit{Client}} := \mathsf{sk}^j_{\mathit{Server}} := A^{x_2} B^{y_2} (C/pw_{\mathit{Server},\mathit{Client}})^{z_2} D^{w_2} F^{x_1} G^{y_1} (I/pw_{\mathit{Client}})^{z_1} J^{w_1}.$$

On the other hand, if $pw_{\mathit{Client}} \neq pw_{\mathit{Server},\mathit{Client}}$ (i.e., as the result of a $\mathsf{Corrupt}$ oracle query), the session keys are computed as

$$\mathsf{sk}^i_{\mathit{Client}} := A^{x_2} B^{y_2} (C/pw_{\mathit{Client}})^{z_2} D^{w_2} F^{x_1} G^{y_1} (I/pw_{\mathit{Client}})^{z_1} J^{w_1}$$
$$\mathsf{sk}^j_{\mathit{Server}} := A^{x_2} B^{y_2} (C/pw_{\mathit{Server},\mathit{Client}})^{z_2} D^{w_2} F^{x_1} G^{y_1} (I/pw_{\mathit{Server},\mathit{Client}})^{z_1} J^{w_1}.$$

(Other aspects of the Execute oracle are handled as before.) Under the DDH assumption, one can show that $|\mathsf{fsAdv}_{A,P_0'}(k) - \mathsf{fsAdv}_{A,P_1}(k)| \le \varepsilon(k)$ for some negligible function $\varepsilon(\cdot)$. Details appear in the full version.

In experiment P_2, upon receiving query $\mathsf{Execute}(\mathit{Client}, i, \mathit{Server}, j)$, the simulator checks whether $pw_{\mathit{Client}} = pw_{\mathit{Server},\mathit{Client}}$. If so, $\mathsf{sk}^i_{\mathit{Client}}$ is chosen randomly from $\mathbb{G}$ and $\mathsf{sk}^j_{\mathit{Server}}$ is set equal to $\mathsf{sk}^i_{\mathit{Client}}$. Otherwise, both $\mathsf{sk}^i_{\mathit{Client}}$ and $\mathsf{sk}^j_{\mathit{Server}}$ are chosen independently at random from $\mathbb{G}$.

Claim. $|\mathsf{fsAdv}_{A,P_1}(k) - \mathsf{fsAdv}_{A,P_2}(k)| \le \varepsilon(k)$ for some negligible function $\varepsilon(\cdot)$.

The claim follows from the negligible statistical difference between the distributions on the adversary's view in the two experiments. The proof when $pw_{\mathit{Client}} = pw_{\mathit{Server},\mathit{Client}}$ exactly parallels the proof given in [14]. When $pw_{\mathit{Client}} \ne pw_{\mathit{Server},\mathit{Client}}$, the proof is more complicated but the techniques used are exactly the same. Details are omitted here and appear in the full version. $\square$

We now introduce some notation. For a query $\mathsf{Send}_1(\mathit{Server}, j, \mathit{msg\text{-}in})$, we say *msg-in* is *previously-used* if it was ever previously output by a Send_0 oracle. Similarly, for a query $\mathsf{Send}_2(\mathit{Client}, i, \mathit{msg\text{-}in})$, we say *msg-in* is *previously-used* if it was ever previously output by a Send_1 oracle. A *msg-in* which is not previously-used is called *new*.

In experiment P_3, the simulator runs the modified initialization procedure shown in Figure 2, where the values $\chi_1, \chi_2, \xi_1, \xi_2, \kappa$ are stored for future use. A

$$
\begin{array}{l}
\mathsf{Initialize}(1^k) \text{ --- } \\
\quad g_1, g_2 \leftarrow \mathbb{G} \\
\quad \chi_1, \chi_2, \xi_1, \xi_2, \kappa \leftarrow \mathbb{Z}_q \\
\quad h := g_1^{\kappa}; \;\; c := g_1^{\chi_1} g_2^{\chi_2}; \;\; d := g_1^{\xi_1} g_2^{\xi_2} \\
\quad H \leftarrow \mathsf{UOWH}(1^k) \\
\quad \textsf{return } \mathbb{G}, g_1, g_2, h, c, d, H
\end{array}
$$

Fig. 2. Modified initialization procedure.

new *msg-in* for a query $\mathsf{Send}_2(\mathit{Client}, i, \langle \mathit{Server}|E|F|G|I|J\rangle)$ is said to be *newly valid*[5] only if $F^{\chi_1+\beta\xi_1} G^{\chi_2+\beta\xi_2} = J$ and $I/pw_{\mathit{Client}} = F^{\kappa}$. Otherwise, we say that it is *newly invalid*. A new *msg-in* for a query $\mathsf{Send}_1(\mathit{Server}, j, \langle \mathit{Client}|\mathsf{VK}|A|B|C|D\rangle)$ is *newly valid* only if $A^{\chi_1+\alpha\xi_1} B^{\chi_2+\alpha\xi_2} = D$ and $C/pw_{\mathit{Server},\mathit{Client}} = A^{\kappa}$, where the value of $pw_{\mathit{Server},\mathit{Client}}$ is that *at the time of the* Send_1 *query* (note that $pw_{\mathit{Server},\mathit{Client}}$ may change as a result of Corrupt queries). Otherwise, we say it is *newly invalid*.

Upon receiving query $\mathsf{Send}_2(\mathit{Client}, i, \mathit{msg\text{-}in})$, the simulator checks whether there has previously been any query of the form $\mathsf{Corrupt}_1(\mathit{Client})$. If so, the Send_2 query is answered as before. Otherwise, the simulator examines *msg-in*. If

[5] Note that (with negligible probability) a newly valid message may not actually be a valid protocol message. This is fine as far as the proof of security is concerned.

msg-in is newly invalid, the query is answered as in experiment P_2. If *msg-in* is newly valid, the query is answered as in experiment P_2 but the simulator stores the value $(\nabla, \mathsf{sk}^i_{Client})$ as the "session key". If *msg-in* is previously-used, the simulator checks for the unique Send_1 query following which *msg-in* was output. Say this query was $\mathsf{Send}_1(Server, j, msg\text{-}in')$. If *msg-in'* is newly invalid, the Send_2 query is answered as in experiment P_2. If *msg-in'* is newly valid, the query is answered as in experiment P_2 but the simulator stores the value $(\nabla, \mathsf{sk}^i_{Client})$ as the "session key".

Upon receiving query $\mathsf{Send}_3(Server, j, msg\text{-}in)$, let $Client \stackrel{\text{def}}{=} \mathsf{pid}^j_{Server}$. The simulator first checks when the query $\mathsf{Send}_1(Server, j, \mathit{first\text{-}msg\text{-}in})$ was made. If this Send_1 query was made after a query $\mathsf{Corrupt}_1(Client)$ or a query of the form $\mathsf{Corrupt}_2(Server, Client, *)$, the behavior of the Send_3 oracle is unchanged. Otherwise, the simulator responds as in experiment P_2 unless *first-msg-in* is newly valid. In this case, the query is answered as in experiment P_2 but the simulator stores the value $(\nabla, \mathsf{sk}^j_{Server})$ as the "session key".

If the adversary queries $\mathsf{Test}(U, i)$ or $\mathsf{Reveal}(U, i)$ before any queries of the form $\mathsf{Corrupt}_1(U)$ or $\mathsf{Corrupt}_2(U, U', *)$ (where $U' = \mathsf{pid}^i_U$), and the session key stored is of the form $(\nabla, \mathsf{sk}^i_U)$, the adversary is given ∇. If the Test or Reveal query is made after any such $\mathsf{Corrupt}$ query has been made, the adversary is given the value sk^i_U. Other Test or Reveal queries are answered as in experiment P_2. Completing the description of experiment P_3, if the adversary ever receives the value ∇ in response to a Reveal or Test query, the adversary succeeds. Otherwise, the adversary succeeds, as before, by correctly guessing the bit b.

Since the adversary's view is unchanged unless it receives the value ∇ and the adversary succeeds when this is the case, we clearly have $\mathsf{fsAdv}_{A,P_2}(k) \leq \mathsf{fsAdv}_{A,P_3}(k)$.

For the remaining transformations, the simulator will modify its actions in response to query $\mathsf{Send}_2(U)$ only when this query is made before a query $\mathsf{Corrupt}_1(U)$. Similarly, the simulator's response to a query $\mathsf{Send}_3(U, i, *)$ (where $\mathsf{pid}^i_U = U'$) will be changed only if the query $\mathsf{Send}_1(U, i, *)$ was made before any queries $\mathsf{Corrupt}_2(U, U', *)$ or $\mathsf{Corrupt}_1(U')$. For brevity in what follows, however, we will simply say that the simulator modifies its behavior only for Send oracle queries made "before any $\mathsf{Corrupt}$ queries".

In experiment P_4, whenever the simulator responds to a Send_2 query it stores the values (K, β, x, y, z, w), where $K = g_1^x g_2^y h^z (cd^\beta)^w$. Upon receiving a query $\mathsf{Send}_3(Server, j, \langle K|\mathsf{Sig}\rangle)$ (assuming query $\mathsf{Send}_1(Server, j, \mathit{first\text{-}msg\text{-}in})$ was made before any $\mathsf{Corrupt}$ queries), the simulator checks the value of $\mathit{first\text{-}msg\text{-}in} = \langle Client|\mathsf{VK}|A|B|C|D\rangle$. If *first-msg-in* is new, the query is answered as in experiment P_3. If *first-msg-in* is previously-used and $\mathsf{Vrfy}_{\mathsf{VK}}(\beta|K, \mathsf{Sig}) = 0$, the query is answered as in experiment P_3 (and the session key is not assigned a value). If *first-msg-in* is previously-used, $\mathsf{Vrfy}_{\mathsf{VK}}(\beta|K, \mathsf{Sig}) = 1$, and the experiment is not aborted, the simulator first checks whether there exists a (unique) i such that $\mathsf{sid}^i_{Client} = \mathsf{sid}^j_{Server}$. If so, sk^j_{Server} is assigned the value sk^i_{Client}. Otherwise, let $\mathit{first\text{-}msg\text{-}out} = \langle Server|E|F|G|I|J\rangle$. The simulator must have stored values x', y', z', w' such that $K = g_1^{x'} g_2^{y'} h^{z'} (cd^\beta)^{w'}$ (this is true since the experiment is aborted if Sig is a valid signature on $\beta|K$ that was not output by the simulator following a Send_2 query). The session key is then assigned the value:

$$\mathsf{sk}^j_{Server} := A^x B^y (C/pw)^z D^w F^{x'} G^{y'} (I/pw)^{z'} J^{w'}.$$

Claim. $\mathsf{fsAdv}_{A,P_4}(k) = \mathsf{fsAdv}_{A,P_3}(k)$.

The distribution on the adversary's view is identical in experiments P_3 and P_4. It is crucial here that the value $pw = pw_{Server,Client}$ used when responding to a Send_1 query is stored as part of the state and thus the same value is used subsequently when responding to a Send_3 query. If this were not the case, the value of $pw_{Server,Client}$ could change as the result of a $\mathsf{Corrupt}_2$ query sometime between the Send_1 and Send_3 queries. □

In experiment P_5, upon receiving a query $\mathsf{Send}_3(Server, j, \langle K|\mathsf{Sig}\rangle)$ (again, assuming query $\mathsf{Send}_1(Server, j, \mathit{first\text{-}msg\text{-}in})$ was made before any $\mathsf{Corrupt}$ queries), the simulator checks the value of *first-msg-in*. If *first-msg-in* is newly invalid and the session key is to be assigned a value, the session key is assigned a value randomly chosen in $\mathbb{G}$. Otherwise, the query is answered as in experiment P_4.

Claim. $\mathsf{fsAdv}_{A,P_5}(k) = \mathsf{fsAdv}_{A,P_4}(k)$.

This step is similar to the identical step in [14], except that it is now crucial that the same value $pw_{Server,Client}$ is used during both the Send_1 and Send_3 queries for a given instance. □

In experiment P_6, upon receiving query $\mathsf{Send}_1(Server, j, \mathit{msg\text{-}in})$ (and assuming this query is made before any $\mathsf{Corrupt}$ queries), if *msg-in* is newly valid the query is answered as before. Otherwise, component I is computed as $h^r g_1^{N+1}$, where the dictionary of legal passwords is $\{1, \ldots, N\}$.

Claim. Under the DDH assumption, $|\mathsf{fsAdv}_{A,P_5}(k) - \mathsf{fsAdv}_{A,P_6}(k)| \le \varepsilon(k)$, for some negligible function $\varepsilon(\cdot)$.

The proof of this claim follows [14], and depends on the non-malleability of messages in the protocol. Details appear in the full version. □

Note that when a query $\mathsf{Send}_1(Server, j, \mathit{first\text{-}msg\text{-}in})$ is made before any $\mathsf{Corrupt}$ queries and *first-msg-in* is not newly valid, the simulator will not require r in order to respond to the (subsequent) query $\mathsf{Send}_3(Server, j, \mathit{msg\text{-}in})$ in case this query is ever made. On the other hand, in contrast to the basic case when no $\mathsf{Corrupt}$ queries are allowed, the simulator *does* require r in case *first-msg-in* is newly valid. The reason is the following: assume the adversary queries $\mathsf{Send}_1(Server, j, \langle Client|\mathsf{VK}|A|B|C|D\rangle)$ (before any $\mathsf{Corrupt}$ queries have been made) where $\langle Client|\mathsf{VK}|A|B|C|D\rangle$ is newly valid. Assume further that the adversary subsequently learns pw_{Client} from a $\mathsf{Corrupt}_1$ query. The adversary might then query $\mathsf{Send}_3(Server, j, \langle K|\mathsf{Sig}\rangle)$ followed by $\mathsf{Reveal}(Server, j)$. In this case, the simulator must give the adversary the correct session key sk^j_{Server} — but this will be impossible without knowledge of r.

In experiment P_7, queries to the Send_2 oracle are handled differently (when they are made before any $\mathsf{Corrupt}$ queries). Whenever *msg-in* is newly invalid, the session key will be assigned a value chosen randomly from $\mathbb{G}$. If *msg-in* is previously-used, the simulator checks for the Send_1 query after which *msg-in* was output. Say this query was $\mathsf{Send}_1(\mathit{Server}, j, \mathit{msg\text{-}in}')$. If $\mathit{msg\text{-}in}'$ is newly valid, the Send_2 query is answered as before. If $\mathit{msg\text{-}in}'$ is newly invalid, the session key is assigned a value chosen randomly from $\mathbb{G}$. Queries $\mathsf{Send}_3(\mathit{Server}, j, \mathit{msg\text{-}in})$ are also handled differently (assuming query $\mathsf{Send}_1(\mathit{Server}, j, \mathit{first\text{-}msg\text{-}in})$ was made before any $\mathsf{Corrupt}$ queries). If *first-msg-in* is previously-used, $\mathsf{Vrfy}_{\mathsf{VK}}(\beta|K, \mathsf{Sig}) = 1$, and there does not exist an i such that $\mathsf{sid}^i_{Client} = \mathsf{sid}^j_{Server}$, then the session key is assigned a value chosen randomly from $\mathbb{G}$.

Claim. $\mathsf{fsAdv}_{A,P_7}(k) = \mathsf{fsAdv}_{A,P_6}(k)$.

The claim follows from the equivalence of the distributions on the adversary's views in the two experiments, as in [14]. Details appear in the full version. □

Let $\mathsf{fsSucc1}$ denote the event that the adversary succeeds by receiving a value ∇ following a Test or Reveal query (this can occur only before any $\mathsf{Corrupt}$ queries have been made). We have:

$$\Pr_{A,P_7}[\mathsf{fsSucc}] = \Pr_{A,P_7}[\mathsf{fsSucc1}] + \Pr_{A,P_7}[\mathsf{fsSucc}|\overline{\mathsf{fsSucc1}}] \cdot \Pr_{A,P_7}[\overline{\mathsf{fsSucc1}}].$$

Event $\mathsf{fsSucc} \wedge \overline{\mathsf{fsSucc1}}$ can occur in one of two ways (by definition of freshness): either (1) the adversary queries $\mathsf{Test}(U, i)$ before any $\mathsf{Corrupt}$ queries and does not receive ∇ in return; or (2) the adversary queries $\mathsf{Test}(U, i)$ after a $\mathsf{Corrupt}$ query and $\mathsf{acc}^i_U = \text{TRUE}$ but the adversary has never queried $\mathsf{Send}_\ell(U, i, M)$ for any ℓ, M. In either case, sk^i_U is randomly chosen from $\mathbb{G}$ independent of the remainder of the experiment; therefore, we have $\Pr_{A,P_7}[\mathsf{fsSucc}|\overline{\mathsf{fsSucc1}}] = 1/2$. Thus, $\Pr_{A,P_7}[\mathsf{fsSucc}] = 1/2 + 1/2 \cdot \Pr_{A,P_7}[\mathsf{fsSucc1}]$. We now upper bound $\Pr_{A,P_7}[\mathsf{fsSucc1}]$.

We define experiment P_7' in which the adversary succeeds only if it receives a value ∇ in response to a Reveal or Test query. Clearly $\Pr_{A,P_7'}[\mathsf{fsSucc}] = \Pr_{A,P_7}[\mathsf{fsSucc1}]$ by definition of event $\mathsf{fsSucc1}$. Since the adversary cannot succeed in experiment P_7' once a $\mathsf{Corrupt}$ query is made, the simulator aborts the experiment if this is ever the case. For this reason, whenever the simulator would previously store values (∇, sk) as a "session key" (with sk being returned only in response to a Test or Reveal query *after* a $\mathsf{Corrupt}$ query), the simulator now need store only ∇.

In experiment P_8, queries to the Send_1 oracle are handled differently. Now, the simulator computes I as $h^r g_1^{N+1}$ even when *msg-in* is newly valid. Following a proof similar to those in [14], it follows under the DDH assumption that, for some negligible function $\varepsilon(\cdot)$, we have $|\mathsf{fsAdv}_{A,P_8}(k) - \mathsf{fsAdv}_{A,P_7'}(k)| \leq \varepsilon(k)$. A key point used in the proof is that when *msg-in* is newly valid, the simulator no longer need worry about simulating the adversary's view following a $\mathsf{Corrupt}$ query.

In experiment P_9, queries to the Send_0 are handled differently. Now, the simulator always computes C as $h^r g_1^{N+1}$. Following a proof similar to those in

[14], it follows under the DDH assumption that, for some negligible function $\varepsilon(\cdot)$, we have $|\mathsf{fsAdv}_{A,P_9}(k) - \mathsf{fsAdv}_{A,P_8}(k)| \leq \varepsilon(k)$. Note that, because session keys computed during a Send_2 query are always either ∇ or are chosen randomly from $\mathbb{G}$, the value r is not needed by the simulator and hence we can use the simulator to break the Cramer-Shoup encryption scheme as in [14].

The adversary's view in experiment P_9 is independent of the passwords chosen by the simulator until the adversary receives ∇ in response to a Reveal or Test query (in which case the adversary succeeds). Thus, the probability that the adversary receives ∇, which is exactly $\Pr_{A,P_9}[\mathsf{fsSucc}]$, is at most $Q(k)/N$. This completes the proof of the theorem.

References

1. M. Bellare, R. Canetti, and H. Krawczyk. A Modular Approach to the Design and Analysis of Authentication and Key Exchange Protocols. STOC '98.
2. M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated Key Exchange Secure Against Dictionary Attacks. Eurocrypt '00.
3. M. Bellare and P. Rogaway. Entity Authentication and Key Distribution. Crypto '93.
4. M. Bellare and P. Rogaway. Provably-Secure Session Key Distribution: the Three Party Case. STOC '95.
5. S.M. Bellovin and M. Merritt. Encrypted Key Exchange: Password-Based Protocols Secure Against Dictionary Attacks. *IEEE Symposium on Research in Security and Privacy*, IEEE, 1992, pp. 72–84.
6. R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, and M. Yung. Systematic Design of Two-Party Authentication Protocols. Crypto '91.
7. M. Boyarsky. Public-Key Cryptography and Password Protocols: The Multi-User Case. ACM CCCS '99.
8. V. Boyko, P. MacKenzie, and S. Patel. Provably-Secure Password-Authenticated Key Exchange Using Diffie-Hellman. Eurocrypt '00.
9. W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6): 644–654 (1976).
10. W. Diffie, P. van Oorschot, and M. Wiener. Authentication and Authenticated Key Exchanges. *Designs, Codes, and Cryptography*, 2(2): 107–125 (1992).
11. O. Goldreich and Y. Lindell. Session-Key Generation Using Human Passwords Only. Crypto '01.
12. S. Halevi and H. Krawczyk. Public-Key Cryptography and Password Protocols. *ACM Transactions on Information and System Security*, 2(3): 230–268 (1999).
13. J. Katz. Efficient Cryptographic Protocols Preventing "Man-in-the-Middle" Attacks. PhD thesis, Columbia University, 2002.
14. J. Katz, R. Ostrovsky, and M. Yung. Efficient Password-Authenticated Key Exchange Using Human-Memorable Passwords. Eurocrypt '01.
15. T.M.A. Lomas, L. Gong, J.H. Saltzer, and R.M. Needham. Reducing Risks from Poorly-Chosen Keys. *ACM Operating Systems Review*, 23(5): 14–18 (1989).
16. P. MacKenzie. More Efficient Password-Authenticated Key Exchange. RSA '01.
17. P. MacKenzie. On the Security of the SPEKE Password-Authenticated Key-Exchange Protocol. Manuscript, 2001.
18. P. MacKenzie. Personal communication. April, 2002.

19. P. MacKenzie, S. Patel, and R. Swaminathan. Password-Authenticated Key Exchange Based on RSA. Asiacrypt '00.
20. V. Shoup. On Formal Models for Secure Key Exchange. Available at `http://eprint.iacr.org/1999/012`.
21. T. Wu. The Secure Remote Password Protocol. *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, 1998, pp. 97–111.