HPAKE : Password Authentication Secure against Cross-Site User Impersonation

Xavier Boyen

Stanford University
xb@cs.stanford.edu

Abstract. We propose a new kind of asymmetric mutual authentication from passwords with stronger privacy against malicious servers, lest they be tempted to engage in "cross-site user impersonation" to each other.

It enables a person to authenticate (with) arbitrarily many independent servers, over adversarial channels, using a memorable and reusable single short password. Beside the usual PAKE security guarantees, our framework goes to lengths to secure the password against brute-force cracking from privileged server information.

1 Introduction

Password-based authentication and key exchange is the process whereby a client achieves mutual authentication with a remote server over an adversarial channel, turning it into a virtual secure communication channel, on the basis of a short password that should be easy to memorize but not guess.

Shared-Password Authentication. (Symmetric) password-authenticated key exchange (PAKE) assumes that the password is shared between the client and the server. The threat in this case is that a (passive or active) outside attacker might try to impersonate either party to the other, or to eavesdrop on the communication taking place within the secure channel. Though such attacks cannot be prevented in an adversarial network, they can be made to require one fresh online authentication attempt for each password being tested. This is a solved problem: many PAKE protocols achieve this notion very efficiently.

Private-Password Authentication. Asymmetric password-authenticated key exchange (APAKE), by contrast, allows the password to be known to the client only. The server holds a long-term authentication token, related in some way to the password, but from which it is (presumably) hard to recover the password itself. In addition to the unavoidable online attack, a secondary threat of concern here is that a compromise of the server database might give to the attacker the means of impersonating its users to another server. Thwarting this threat means that it is safe for a client to to reuse the same password with multiple servers. This constitutes a very significant usabily improvement around human limitations.

J.A. Garay, A. Miyaji, and A. Otsuka (Eds.): CANS 2009, LNCS 5888, pp. 279–298, 2009.

Many elegant APAKE protocols have been proposed over the years, that deliver more or less optimally on all those requirements — provided that the password is not too weak, that is. Indeed, a general attack strategy for the evil insider in the APAKE model is, once it has obtained the server's database, to mount an offline dictionary attack to recover the client password: a server can always do that, simply by simulating the authentication protocol with itself posing as the target user. Since such attack cannot be prevented, the user's only recourse is to make the attack slower, which requires: (1) that the protocol itself be made intentionally slower; (2) and that the server implements it correctly. Both are undesirable requirements.

On Password Strength. We remark in passing that the threshold for what constitutes a "good" password is *much* higher in the insider threat model than the outsider one, even though there are many more potential outsider attackers. Online attacks, the only option for outsiders, are indeed inherently slow and can be artificially and arbitrarily rate-limited; they are also easy to detect and counter by locking up the account.

Thus, as long as some basic security requirements are met against outsider attacks, one is much better served by devoting one's energy to the prevention of insider attacks.

Misaligned Incentives. Unfortunately for the end-user, servers generally have little incentive to assist in this task, since (1) it would presumably make the protocol more costly, and (2) it would be giving into the suggestion they, the servers, cannot be trusted with the users' passwords.

At a fundamental level, the entity most harmed in case of a password breach would be the owner of the account, not the server providing the service. The user thus has a greater incentive to do something about this, e.g., accept a slower protocols if it can make the password safer. Alas, users generally have no power to dictate such a change. The only available option is generally to preprocess the password *outside* of the protocol before it starts. (Though better than nothing, a problem of this approach is that the preprocessing function must be non-parametric if one is not willing to accept a statefull client on which the parameters can be stored).

Equally worrisome, the APAKE model does not explicitly take into account the threat of *cross-site user impersonation*, where the server itself at site A turns rogue and attempts to impersonate the client at some other site B (based on the oft-fulfilled premise that the client picked identical or very similar passwords on both sites). Since Server A *itself* is the threat in this attack scenario, one cannot reasonably expect it willingly to fight against itself (unless an external mechanism such as a reputation at stake comes into play). It could also be a matter of denial; after all, website operators who are genuinely honest will most likely consider themselves trustworthy — regardless of whether the user trusts them or not.

Reinternalizing the Externality. In economic parlance, one would say that, in the insider threat model, the incentives of the parties are mis-aligned; their

wishes are at odds with each other. The root of the problem is that there an economic *externality*: the client is the one who suffers if the server fails to protect its database adequately.

(One of) our goal is thus to design a protocol whose "pricing structure" re-internalizes this externality. But, first, we look at other simpler alternatives. (We only consider alternatives that require no custom data storage, secret or public, on the client side; with it, our problems would be solved.)

Client-side Preprocessing. The application of a complex transformation on the password, *e.g.*, hashing it many times before use, is the implicit customary defense against offline server threats. Preprocessing can be very useful, if done by the client, because it realigns the costs with the incentives. However, it also creates new problems of its own, depending whether its complexity is fixed or variable.

Fixed-cost preprocessing, *e.g.*, with a hash function or password-based key derivation function of fixed constant complexity, is easy to implement, but it is a rather blunt instrument that can be too slow and inconvenient in some situations, and not provide enough of a deterrent against attacks in others.

Parametric-cost preprocessing, *i.e.*, based upon a user-selected complexity parameter, poses another problem, which is that the parameter must be stored somewhere, and available for the client to retrieve whenever needed. [1]

The need to retrieve parameters is what makes parametric preprocessing problematic in practice, because it is generally not desirable to keep them in the clear, and give them to anyone who asks. Indeed, the user's choice of complexity parameter can itself provide very valuable information, e.g, to guide an attack toward a promising target. And hiding the parameter behind an extra layer of authentication is a circular non-solution that just moves the problem around.

A Host of Requirements. As we said, our goal is to realize a secure and "economically sound" password protocol, i.e., with all the usual APAKE security guarantees, plus a provision for the user to defend her password against dishonest servers the way she sees fit. Hence there must be a (secret, user-programmed, user-computed) "computational bottleneck" somewhere that renders insider offline attacks arbitrarily slower, but that does not penalize a honest server.

Intuitively, our "computational bottleneck", or programmable costly function, will have to satisfy the following requirements:

Client-owned bottleneck: As discussed, the only way to thwart offline insider attacks is to make the protocol slower, somewhere. The client must own this feature, since his or her interests are at stake.

Server-side independence: Not only should the server be oblivious to the selection of the client-side bottleneck, it should also be removed from its calculation, for obvious scalability reasons (whereas a human will authenticate

[1] The password authentication system of [42] is based precisely on that idea. It relies on a third-party central server for storing and recalling the cost parameter. Anyone can request and obtain this data.

to one site at a time, a machine may have to answer thousands such requests per second).

Cacheable preprocessing: Because the hard function may, by user choice, take a very large amount of time to compute, it would be nice if the result could be cached in secure storage, for future uses with the same server, for as long as the user deems it safe to keep it there.[2] This requires: (1) the output of the bottleneck function to be independent of any random ephemeral used in the protocol execution; (2) the authentication process to be "hot-started" at the point where the client has just finished evaluating the bottleneck function.

Zero client storage: Conversely, no persistent storage whatsoever should be required of the client. Especially, all secrets should fit in the user's mind (those being the password and nothing else). We specifically demand that the user be allowed to forget the value of the cost parameter once it has been programmed into the registration data sent to the server.

Secrecy of the parameter: In general, in security it is a good idea not to leak any information that is not explicitly needed, unless one can prove that such leak is benign. Leaking the cost parameter is certainly not benign, since it might tell what the important targets are, and reveal other password usage pattern of the user.

Secrecy of the parameter's retrieval: Allowing an attacker to learn the bottleneck parameter can be very damaging, but even more so to let it learn whether the user has recovered it correctly. Depending on the leakage mechanism, e.g., if it comes from the protocol itself, then a dishonest server could use it to mount an offline dictionary attack that entirely bypasses the hard function. Neither party should learn whether a retrieval attempt succeeded, before the protocol actually reaches the accepting state.

Contribution. To address all the issues we raised, we propose the notion of Hardened Password-Authenticated Key Exchange (HPAKE), which integrates a user-programmable hard function with the above properties into an authentication protocol with PAKE and APAKE security.

With it, users will thus be able to reuse the same passwords at various sites, without having to trust that the server or the network is behaving nicely. The benefits over existing solutions, such as APAKE are especially pronounced in the case of weak passwords that would be easy to crack were they used in a regular APAKE protocol.

The architecture of HPAKE is easy to explain generically; it is based on three existing cryptographic primitives used as black boxes; all of them in fact have been known since the dawn of cryptography, except for the preferred instantiation of the user-programmable hard function which is recent.

[2] It is indeed a good strategy for long-term passwords of last resort to pick them very memorable, and thus very weak, and rely on a very high cost parameter to defeat offline attacks.

Setup Assumption. Before proceeding, we should mention that there are a lot of ways to attack password-based systems, many of which do not depend on the protocol used (key loggers and social engineering attacks being two common examples). Our objective is to provide the highest level of security for multi-site password authentication, under the common-sense assumption that neither the human user nor the electronic device acting as the client on his or her behalf (and on which the password will be seized), leaks any information to the outside world other than as specified by the protocol.

On the other hand, we stress once again that we make none of the following all-too-common assumptions: a public-key infrastructure (PKI), a preexisting one-sided authentication mechanism (such as SSL with a root CA), a client-side data storage device of any kind (whether private and/or authentic or neither), or any tamper-proof client hardware that has somehow become tied to the user (including physically unclonable functions or PUF).

2 Related Work

All password-based remote authentication and key exchange protocols can be divided into two broad categories, depending on the nature of the secrets held by the client and the server:

A. Shared-password authentication, where both parties share the same secret. Since there is no password privacy there is no possibility of password reuse. For completeness, we mention:
- cleartext passwords, even if transmitted over an encrypted link *à la* SSL;
- symmetric challenge-response authentication using nonces and hashes;
- various *ad hoc* password-only protocols using public-key techniques;
- most cryptographic password-authenticated key exchange PAKE protocols (see below).

B. Password-private authentication, where the secrets are asymmetric. The client proves possession of the password to a server that proves knowledge of a derived secret. There are:

Stateful schemes, whose clients keep state or carry custom data beside the password, *e.g.*:
- preregistered public keys, where the password unlocks a signing key;
- multi-factor systems, *e.g.*, involving biometric or hardware credentials;
- client-side "password managers" unlocked by a meta-password;
- any authentication system that uses lists of one-time credentials.

Stateless schemes, where the only client custom data is a small secret password. Such protocols truly enable "untethered" roaming for a human user. The only examples are:
- augmented password-authenticated key exchange (APAKE, see below);
- our HPAKE protocol, which is better hardened against malicious servers.

AKE. Key exchange (or key agreement) protocols from high-entropy secrets date back from the original Diffie-Hellman protocol [19]. Authenticated key

exchange (AKE) further ensures that the two parties are mutually authenticated, *i.e.*, that they have the proper long-term secrets, and thus that no impersonation is taking place. Since achieving AKE from a shared high-entropy secret is all but trivial, mentions of AKE in the literature truly refer to "asymmetic" authentication (AAKE), where each party has its own private secret and has registered the corresponding public key with the other party. This notion of AAKE has been progressively refined and perfected over the years; see for example [20,3,7,15,32,30,37,17,31]. (We make the distinction between AKE and AAKE to emphasize the fact that later on we may elect to use one or the other.)

PAKE. For low-entropy human-memorable secrets, the grandfather of PAKE protocols is the Encrypted Key Exchange (EKE) scheme proposed by Bellovin and Merritt [4], and which can arguably be traced further back to the notion of Privacy Amplification [6]. In both cases, the goal was to take a short shared secret, and boost it into a cryptographically strong one by a public discussion process over an open channel [36]. The EKE protocol provided a particularly efficient way to do so, with (implicit) mutual authentication of the parties. It also jump-started a fruitful line of research, which led to many results including new definitions [3,1,25], increased efficiency [26,33], and/or provable security properties [11,2]. More recently, there has been a surge of interest in the construction of PAKE protocols with better proofs of security that avoid the random-oracle model, *e.g.*, in favor of the common reference string model; we mention the first reasonably efficient such protocol [29,13], and a simpler and faster variant [27]. Although by far most of the constructions are based on a Discrete Log assumption such as Diffie-Hellman or variations thereof, there are protocols based on the RSA assumption [34] or the Phi-hiding assumption [23].

APAKE. Although the EKE protocol of Bellovin and Merritt originally required both parties to know the password, it was soon followed by an asymmetric version called "Augmented" EKE, by the same authors [5], who had realized the impracticality of requiring users to remember independent passwords for different environments. However, it is not until much later that this concern has been addressed again, first in [25] under somewhat stringent operating conditions, then more practically in [2] and in a sequence of papers [11,33] which culminated in the so-called Omega-method [24] for "augmenting" any given symmetric PAKE protocol. Another way to deal with the threat of server corruption and password exposure is to use multiple servers in a threshold scheme, which is the solution adopted in [35], though this requires the user to believe that the servers are not colluding.

KDFs. Many approaches have been proposed to address the problem of offline dictionary attacks, whether for static storage, or in the context of an authentication protocol. Most of these proposals involve the use of password alternatives which are supposedly harder to brute-force without human assistance; we mention the interactive grid-like password system PassMaze [12], schemes based on visual recognition [39], sequences of challenges and responses [40], and solutions

to "captcha"-like problems that are far easier for humans to solve than for computers [14]. In the context of traditional alphanumeric passwords, the method of choice to thwart guessing attacks remains the deliberately slow key derivation functions in the original Unix password log-on, made programmable in [41], and perfected into the secretly user-programmable halting key derivation functions of [9]. These (H)KDFs are somewhat related to the proofs of work used in other contexts [21,28,18].

3 Architecture

The generic HPAKE protocol is shown on Figure 1. We now informally explain what it does. In Section 4 we give more details on its components.

General Overview. In our system, the user and the server hold asymmetric credentials to authenticate each other: for the user, it is her password, for the server, it is a long-term authentication token obtained from the user when she initially registered. The user selects the cost parameter associated with that password/token pair during the initial registration with the server. The password is concealed from the server, and so is the cost parameter (see below). Once the registration is completed, the user can forget everything (*e.g.*, the token given to the server, and the cost parameter) except the password.

Later, when the user wishes to establish a secure session with the server, she sends a (blind) commitment to the server. The server responds with some ciphertext that depends on the commitment. The client uses some of that ciphertext as input to the hard function, and performs the computation (which may take a while). If she committed to the correct password, the hard function output will let her decrypt the rest of the ciphertext into a copy of the long-term authentication token held by the server. Based on this, the two parties can then mutually authenticate each other and set up a secure channel.

User Programmability and Parameter Secrecy. There are good reasons for letting the user select the complexity parameter associated with his password; but it is equally important to prevent anyone from learning this value prematurely (i.e., not until they have successfully completed the authentication).

At the same time, such value must be stored somewhere, since we cannot ask the user to remember it from memory (the only thing he should be asked to remember being the password).

This requirement of a user-programmable computational bottleneck whose cost parameter is hidden from everyone and yet implicitly stored, *requires* a specific kind of unpredictable function: one that halts (after the prescribed cost expenditure) *only* on the correct input — and that on all other inputs proceeds indefinitely without ever giving back any hint that its input might have been wrong. We refer to such functions as "(selectively) halting functions".

Selectively Halting Functions. Such notion of halting function is closely related to that of Halting Key Derivation Function (HKDF) used in [9] to boost the

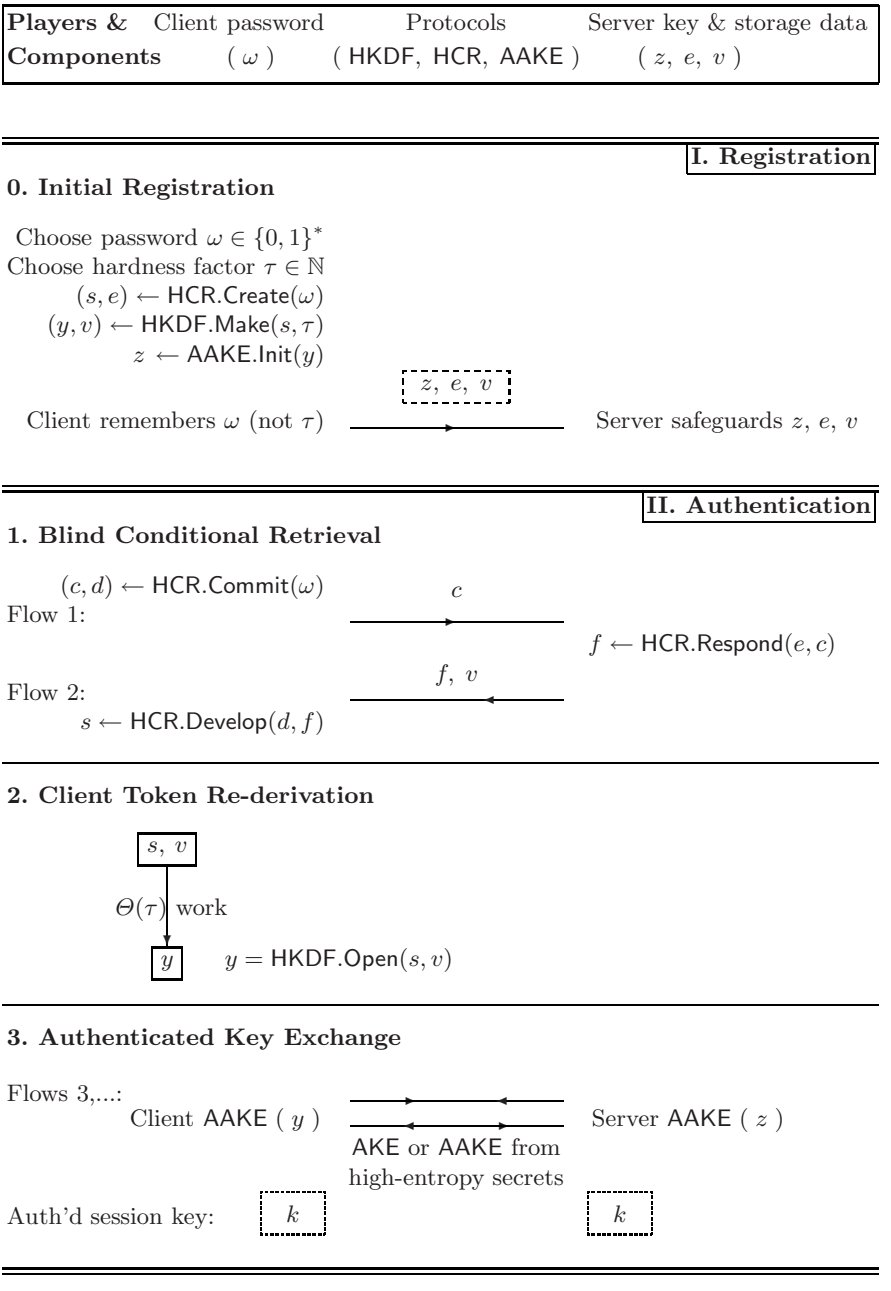

Fig. 1. The generic HPAKE protocol

security of stand-alone password-based encryption. Indeed, conditionally halting functions such as HKDFs have another surprising benefit (which was the main point of [9]): they provide more security than any key derivation function KDF whose computational cost is known, for the same cost and the same password.

(Precisely, it is shown in [9] that a game-theoretically optimal attacker who has no idea about the programmed cost parameter must expend about 3.59× more work than if it knew it, *e.g.*, if it were facing a regular KDF with an explicit parameter.)

Incorporating stand-alone Halting Key Derivation Functions (HKDF) into our two-party key exchange protocol requires some precautions, because we want the *client* to compute it, but the *server* to store most of its input (since the client is memory-constrained, and the server mostly time-constrained). A fundamental and unavoidable problem with HKDFs is that they can serve as a password test predicate, since by definition they halt only on the correct input, which is a testable behavior. The consequence is that we will need a way to transport that data from server to client without exposing it to outside attackers, with the main complication being that the client will not have been authenticated yet by the time it needs the HKDF data.

Security by Obscurity? We emphatically stress that this notion of concealing a secretly programmed cost parameter from the adversary is not "security by obscurity", because all parties are deprived of the secret parameter, including the user who may safely forget the choice once it has been made and registered.

4 Components

We now give more details on the three cryptographic functions used in HPAKE.

Secure Registration. We note that the registration phase is special and not truly part of the protocol. It requires a secure channel which can stem from a face-to-face meeting or from a trusted PKI (which need not be used again in the actual protocol execution). Registration exists so that a user and a server who have never been in contact can start somewhere.

4.1 HKDF : Halting Key Derivation Functions

"Halting Key Derivation Functions" were originally defined in [9] to derive strong keys from weak passwords in a rate-limiting manner, to be used in a stand-alone password-based encryption system.

Here, we use HKDFs slightly differently: to map one secret random string (the retrieved secret s) into another (the client-side token y), in a manner that can be made as computationally expensive as the user wishes by selecting a suitable value of the parameter τ.

The primitive consists of two algorithms, HKDF.Make and HKDF.Open:

HKDF.Make takes as input a secret $s \in S$, a parameter $\tau \in \mathbb{N}$, and random coins, and returns a random token $y \in Y$ and its ciphertext $v \in V$.

HKDF.Open takes as input a secret $s \in S$ and a ciphertext $v \in V$, and, either returns a token $y \in Y$, or fails to halt in polynomial time.

We briefly recall the security requirements from [9]. For a random execution of Make, it must be infeasible to find, in polynomial time in the security parameter, a tuple (s', s, τ, y) such that $y = \mathsf{Open}(s', \mathsf{Make}(s, \tau))$ and $s \neq s'$. Furthermore, finding a tuple (s, τ, y) such that $y = \mathsf{Open}(s, \mathsf{Make}(s, \tau))$ must require $\Theta(\tau)$ units of time and memory, barring which no information about the correct y must be obtained from v, s, τ.

For concreteness, we give an HKDF construction adapted from [9].

$\mathsf{HKDF.Make} : (s, \tau) \mapsto (v, k)$

$r \leftarrow \{0,1\}^\ell$
$z \leftarrow \mathsf{Hash}(s, r)$
FOR $i := 1, ..., \tau$ or UNTIL user signal
  $z_i \leftarrow z$
  REPEAT q times
    $j \leftarrow 1 + (z \bmod i)$
    $z \leftarrow \mathsf{Hash}(z, z_j)$
$v \leftarrow (r, \mathsf{Hash}(z_1, z))$
$k \leftarrow \mathsf{Hash}(z, r)$

$\mathsf{HKDF.Open} : (s, v) \mapsto k$

parse v as (r, h)
$z \leftarrow \mathsf{Hash}(s, r)$
FOR $i := 1, ..., \infty$
  $z_i \leftarrow z$
  REPEAT q times
    $j \leftarrow 1 + (z \bmod i)$
    $z \leftarrow \mathsf{Hash}(z, z_j)$
  IF $\mathsf{Hash}(z_1, z) = h$ BREAK
$k \leftarrow \mathsf{Hash}(z, r)$

The constant q is a design parameter that determines the ratio between the time and space requirements. It is not critical and wide range of values are acceptable for this parameter [9].

The primary purpose of using HKDFs is to let the user impose a computational cost without revealing it to the server or storing extra parameters locally.

The secondary benefit of HKDFs is that they are always at least as difficult to crack as a regular KDF of equal computational cost, and usually more depending on how wide or how far of the attacker's distribution of τ is compared to the user's choice.

We refer the reader to [9] for a full analysis and explanation of these phenomena. Suffice it to say that, in the best case, HKDFs provide a constant security multiplier of 3.59 (or 1.84 bits) over comparable KDFs, and in the worst case the multiplier is 1 (or 0 bit). In other words, HKDFs are never worse, and usually better than regular KDFs of same cost. To reap those benefits, the user-selected cost parameter must not be known exactly to the attacker, which is why it is important to let the user choose it, perhaps haphazardly, on a case-by-case basis.

4.2 HCR: Hidden Credential Retrieval

"Hidden Credential Retrieval" [10], the next ingredient, is a very simple cryptographic abstraction that allows a stateless client to retrieve some high-entropy

secret s from a ciphertext e on remote storage server, based on a low-entropy password ω, in the safest possible way over an insecure channel. A feature of HCR is that it also protects the user data s and password ω against a curious server: the server only has in its custody a blinded string or ciphertext e, from which it is information-theoretically impossible to recognize either s or ω without also knowing the other. Furthermore, no party is to learn from the HCR protocol whether the user successfully retrieved the string s: in case of incorrect password, a junk string is silently recovered instead.

HCR was first formalized and utilized in [10] as a stand-alone protocol, though similar notions have been implicitly proposed much earlier, in different contexts. Notably, the notion of blind signature, coupled to some mild additional conditions (single-round signing and uniqueness of the unblinded signatures) already fulfilled in Chaum's original paper [16], subsumes that of HCR.

To define it, we consider three entities: a Preparer $\mathcal{P}$ that selects the retrieval password ω and the random string s to be stored; a Querier $\mathcal{Q}$ that knows the password ω and seeks to retrieve s; and a Responder $\mathcal{R}$ that acts as the storage server, prepared by the preparer and responding to queries from the querier. Both $\mathcal{P}$ and $\mathcal{Q}$ are meant to embody the same user, but we must separate the two to account for the possibility that the user does not need to remember s once it has finished to set up the server $\mathcal{R}$. The protocol consists of four algorithms:

- HCR.Create, used by the Preparer $\mathcal{P}$, takes as input a reference password ω, and ouputs a plaintext s and a ciphertext e. The plaintext and ciphertext have uniform marginal distributions in some fixed sets S and Z respectively (that is, both s and e are marginally, but not jointly, independent of ω).
- HCR.Commit, used by the Querier $\mathcal{Q}$, takes as input a query password ω, and outputs a commitment c and some private information d. The commitment is uniform in some set C and statistically independent of the query password.
- HCR.Respond, used by the Responder $\mathcal{R}$, takes as input a ciphertext e and a commitment c, and outputs a response f in some set F.
- HCR.Develop, used by the Querier $\mathcal{Q}$, takes as input the private data d and the response f, and outputs a plaintext s in the set S.

We refer to [10] for the formal security model of this primitive, and the various ways to construct it, but note that HCR can be constructed immediately from (very old) existing constructions such as unique blind signatures, including Chaum's [16] and Boldyreva's [8]. The Ford-Kalisky server-assisted password generation protocol from [22] is also an instantiation of HCR (though the use that Ford and Kalisky proposed for their scheme was different).

For illustration purposes, we describe the Ford-Kalisky version which is a bit simpler, but the Boldyreva signature would do just as well. Let $\mathbb{G}$ be a cyclic group of prime order p, and let $\mathsf{Hash} : \{0,1\}^* \to \mathbb{G}$ be a hash function into $\mathbb{G}$.

- HCR.Create : $\omega \mapsto (e, s)$. On input a registration password $\omega \in \{0,1\}^*$, output a storage-server string $e \in_\$ \mathbb{F}_p^\times$ and a user plaintext $s \leftarrow \mathsf{Hash}(\omega)^e$.
- HCR.Commit : $\omega \mapsto (c, d)$. Given any candidate password $\omega \in \{0,1\}^*$, output a private decommitment $d \in_\$ \mathbb{F}_p^\times$ and a public commitment $c \leftarrow \mathsf{Hash}(\omega)^d$.

$\mathsf{HCR.Respond} : (e, c) \mapsto f$. Given the ciphertext $e \in \mathbb{F}_p^\times$ and a commitment $c \in \mathbb{G}$, output the (deterministic) blind response $f \leftarrow c^e$.

$\mathsf{HCR.Develop} : (d, f) \mapsto s$. Given an ephemeral $d \in \mathbb{F}_p^\times$ and a response $f \in \mathbb{G}$, output the retrieved (but unverified) user plaintext $s \leftarrow f^{1/d}$.

4.3 AKE: Authenticated Key Exchange

"(Asymmetric) Authenticated Key Exchange" is our third and final ingredient. Although it may seem strange to require an (A)AKE to build an HPAKE, there is no circularity given that AKE or AAKE from high-entropy keys is quite easy and very well known. In our description, the AKE shared secret, or the AAKE conjugate secrets, are the client-side token y and the server-side token z (such that $y = z$ for AKE or $y \neq z$ for AAKE, respectively).

Choosing an AAKE scheme for this stage instead of AKE (*i.e.*, with asymmetric secrets), will result in resistance to the compromise of the server database, even for authentication to the same server. That is, even with knowledge of all the server secrets including z, impersonating the client to the server itself will still require finding the password (and thus cracking the HKDF). The AAKE server token is initialized at registration time by the client; we generically wrote $z = \mathsf{AAKE.Init}(y)$, but in practice y and z will be returned together by a key generation algorithm. Efficient AAKE schemes include [15] or the very compact MQV [32] on elliptic curves.

Alternatively, for increased server-side efficiency it is possible to use a symmetric AKE scheme instead. In this case, the client and server tokens are the same: $z = y$, though they will still vary from one server to the next even under the same password. We this choice, we forgo resistance to server database compromise against the same server, but we still get all the other security properties of HPAKE, including password secrecy and resistance to cross-site impersonation attacks. (Indeed, an attacker who learns $y = z$ for a specific client-server pair will be able to impersonate that client to that same server, but not to any other server, and without learning the password.)

For concreteness, we give an explicit "folklore" symmetric AKE protocol built purely from hash functions modeled as random oracles [38]. It is a very efficient three-flow AKE protocol where the client and server send each other fresh random *nonces* n_c and n_s, and verify their correct reception and create a session key by hashing them with the secret key $y = z$ they share.

$\mathsf{AKE}.1 : C \rightarrow S$

C picks a fresh random nonce c_c and sends it to S:

$\mathsf{AKE}.2 : C \leftarrow S$

Using S's stored copy of y and the received values $\hat{c_c}$ of c_c, S sends a fresh random nonce c_s and the value $a_s \leftarrow \mathsf{Hash}(y, c_c, c_s)$ to C.

$\mathsf{AKE}.3 : C \rightarrow S$

Using C's reconstructed copy $\hat{y}$ of y and the received values $\hat{c_s}$ and $\hat{a_s}$ of c_s and a_s, C verifies the equality, $\hat{a_s} \stackrel{?}{=} \mathsf{Hash}(\hat{y}, c_c, \hat{c_s})$. If true, C accepts the session and sends $a_c \leftarrow \mathsf{Hash}(y, c_s, a_s)$ to the server.

AKE : session key

Using S's stored copy of y and the received value $\hat{a_c}$ of a_c, S verifies that, $\hat{a_c} \stackrel{?}{=} \mathsf{Hash}(y, c_s, a_s)$. If true, C accepts the session. At this point, if both parties have accepted, they share a mutually authenticated random session key given by, $k \leftarrow \mathsf{Hash}(y, \hat{a_c}, a_s) = \mathsf{Hash}(\hat{y}, a_c, \hat{a_s})$.

4.4 Consolidation of Flows

The above AKE protocol requires three flows (or half-rounds). If we add the two flows for HCR, that makes five flows for the complete HPAKE protocol.

However, it is possible and easy to interleave and consolidate the HCR and AKE messages so that HPAKE as a whole only requires three flows.

The idea is for the client eagerly to send Flow 1 of AKE along with the HCR commitment c in Phase 1 of HPAKE. The server then sends Flow 2 of AKE along with the HCR response f back to the client. The client performs the HKDF hard-function calculation in Phase 2 of HPAKE, and, once the token y is decrypted, sends the final Flow 3 of AKE, thereby completing Phase 3 of HPAKE in one additional flow instead of three.

It is easy to see that the first two flows of AKE are independent of the HCR phases on the protocol. In the random-oracle model, it is even acceptable to reuse the HCR commitment c directly as the AKE client nonce c_c, thereby saving a little extra bit of bandwidth.

The only drawback of this flow consolidation is that the server needs to preserve the AKE state across Phases 1–3 of the full protocol, while Phase 2 may by design take a long time for the client to complete (unless the client is caching a copy of y, which is explicitly allowed). By contrast, in the plain unconsolidated protocol, the server can remain stateless until Phase 3.

5 Security

Theorem 1. *Let χ be a security parameter, such that all hash functions have at least $\ell \geq 2\chi$ bits of output in the random-oracle model. Let $|\mathcal{D}| \ll 2^{\chi}$ be the size of the password dictionary. Assume that the HCR and AKE subprotocols are χ-bit secure in $\mathbb{G}$, that is, they yield to PPT computational adversaries with time-advantage product $TA \geq 2^{\chi}$ only. Suppose that $|\mathcal{D}| \ll 2^{\chi}$, i.e., the password dictionary size is the weak link. Then, in the random-oracle model, the advantage* Adv *of an polynomial-time adversary $\mathcal{A}$ at distinguishing from random a secure channel established by uncorrupted parties (either by causing one party to accept a new session with $\mathcal{A}$, or by stealing an already established session), is, $\forall k \in \mathbb{N}$:*

- *For an outsider $\mathcal{A}$ sending a total of q messages to the user and any number of honest servers:*
$$\mathsf{Adv}_{\mathcal{A}} \leq \frac{q}{|\mathcal{D}|} + \mathrm{o}(1/\chi^k) \ .$$
- *For an insider $\mathcal{A}$ sending a total of q messages to the user and any number of honest servers, and making t queries to the random oracle used in the*

function HKDF.Open *(expressed in the same unit as the hardness parameter* τ *used in* HKDF.Make *when registering with the insider):*

$$\mathsf{Adv}_{\mathcal{A}} \leq \frac{2t}{|\mathcal{D}|\,\tau} + \frac{q}{|\mathcal{D}|} + \mathrm{o}(1/\chi^k) \qquad \textit{in the general case ;}$$

$$\mathsf{Adv}_{\mathcal{A}} \leq \frac{2t}{3.59\,|\mathcal{D}|\,\tau} + \frac{q}{|\mathcal{D}|} + \mathrm{o}(1/\chi^k) \qquad \textit{in cases where :}$$

- *the amount of memory available to* $\mathcal{A}$ *is* $\leq \mathrm{o}(|\mathcal{D}|\,\tau)$ *(which is always true in practice by a wide margin); and,*
- *either, the parameter* τ *is drawn by the user from a distribution of density* $\sim \tau^{-1-\epsilon}$*, or, the attacker* $\mathcal{A}$ *believes that it is not in its interest to try to guess* τ *(which is generally the case by a game-theoretic argument, see [9] for details).*

5.1 Interpretation

Theorem 1 expresses two very different bounds, depending on whether the user and server(s) are together facing a third-party attacker, or whether the user is facing a malicious server.

Against Outsiders. The advantage of outsiders, $\frac{q}{|\mathcal{D}|} + \mathrm{o}(1/\chi^k)$, is the usual bound for PAKE and APAKE protocols. It corresponds (up to a negligible term) to the unavoidable online attack where the outsider tries to impersonate the user to the server (or *vice versa*) by trying out one password candidate at a time.

Since q is the number of *online* queries, and thus necessarily quite small, the security margin against outsiders will remain acceptable even for very small dictionaries $\mathcal{D}$, and thus very weak passwords. The banking industry, for example, is content to protect user accounts with four-digit PINs, thus with just 13 bits of entropy, by locking the account after three incorrect attempts.

Against Insiders. The advantage of insiders (*i.e.*, corrupt servers) is the same as outsiders plus an additional term, $\frac{2t}{B\,|\mathcal{D}|\,\tau}$, that accounts for the possibility that insiders have to mount an offline attack against the password. Here, τ if the user-selected complexity parameter, and B is a small constant ($B = 1$ or $B = 3.59$ depending on whether the "halting principle" is not, or is, applicable [9], i.e., whether τ is adequately uncertain to the attacker). Together, the dictionary size $|\mathcal{D}|$ and the user-programmable complexity parameter τ constitute the main actionable defenses at the user's disposal to thwart an insider offline attack. (Having $B > 1$ is merely a useful side-effect of enforcing the secrecy of τ, though the latter is already desirable in itself, as discussed previously.)

The offline attack, though it requires insider knowledge to be feasible, is far more dangerous than the online attack already available to outsiders. It is dangerous because, in an offline attack, the numerator t is out of the control of the user (or any honest server). It depends only on the adversary's resources and can therefore be quite large; in particular, $t \gg q$.

It is useful to take a very concrete example to illustrate this point. Suppose that the user, Alice, has a single password, which she uses everywhere, and changes every four months (10^7 seconds). Suppose also that one of the web sites where she has an account is a sham, and wishes to dedicate an enterprise-class computer farm (10^5 CPUs) to the single task of attempting to recover Alice's password. The attacker thus has a window of $10^{12} \approx 2^{40}$ CPU-seconds at his disposal before the password becomes useless. For comparison, Alice's password will succumb with probability $p = \frac{1}{2}$, in each of the following five scenarios ($|\mathcal{D}|$ = dictionary size; τ = user-selected hash complexity):

A. $|\mathcal{D}| = 2^{61}$ (61 bits) and fixed $\tau = 2^{-21}$ ($0.5\mu s$):
i.e., a strong password (13 random letters) with a computer-instantaneous hash (*e.g.*, SHA1);
B. $|\mathcal{D}| = 2^{51}$ (51 bits) and fixed $\tau = 2^{-11}$ ($500\mu s$):
i.e., a strong password (11 random letters) with a number-theoretic hash (*e.g.*, on curves);
C. $|\mathcal{D}| = 2^{38}$ (38 bits) and public parameter $\tau = 2^2$ ($4s$):
i.e., an ok password (8 random letters) with a human-noticeable hash (such as a 4-sec KDF);
D. $|\mathcal{D}| = 2^{38}$ (38 bits) and *secret* user-selected $\tau = 2^0$ ($1s$):
i.e., same password (8 random letters) with a human-instantaneous hash (here, 1-sec HKDF);
E. $|\mathcal{D}| = 2^{24}$ (24 bits) and *secret* user-selected $\tau = 2^{14}$ ($5h$):
i.e., a very memorable but very weak "backup" password (5 random letters) protected by a very expensive hash (5-hours HKDF, taking, *e.g.*, 17 minutes to compute on a 16-core client).

(In all scenarios, the password lengths are for lowercase-only random letters, *i.e.*, a 26-symbol alphabet.)

Case A corresponds to the practice of simply hashing the password (possibly with some site-dependent non-secret information) before use, in a regular PAKE protocol.

Case B corresponds to most AEKE and APAKE protocol implementations, where the KDF is an inherent part of the protocol, and subject to number-theoretical constraints (such as compatibility with efficiently verifiable zero-knowledge proofs of knowledge of the password).

Cases C and D correspond to our HPAKE protocol with everyday settings, where the difference between the two is that in the former the hardness factor τ is a known parameter of the system, while in the latter it is chosen by the user in a somewhat unpredictable way (to the adversary).

Case E corresponds to the use of HPAKE with a last-resort backup password that ought never to be used, but must be very easy to remember in case it is ever needed, for instance because the user forgot her regular password. Because highly memorable passwords are also easy to guess, a very large value of τ is desirable to maintain a sufficient margin of security against insider attacks. (How large τ should be, depends on the actual strenght of the backup password, which is known to the user only. This case illustrates why τ must be kept secret.)

Those examples clearly show the superiority of HPAKE over previous PAKE and APAKE protocols in that it allows much weaker passwords to be safely reused, both in an everyday situation (*e.g.*, comparing Case D *vs.* Case B), as well as in a last-resort backup situation (for which none of the existing protocols offers a viable solution).

6 User Interface

Provided that text passwords are used, the client-side user interface (UI) does not require any special hardware: a keypad is all that is required, with perhaps a one-bit display to indicate that the hard function computation is in progress. There are however two important software requirements:

6.1 Trusted Local Password Entry

All the precautions we took to protect the user password and ensure its reusability are moot if an attacker ever manages to bypass the HPAKE protocol, *e.g.*, by tricking the user into entering the password directly into web form.

Software Solutions. A software solution, specific to internet transactions, would require native HPAKE support from the browser, and ideally from the operating system, so that password-entry prompts can be made distinctive enough to be easy to recognize as genuine by the user. *E.g.*, some browsers already attempt to make HTTP-Auth password dialog boxes look unlike regular browser windows; and the Windows operating system requires a Ctrl-Alt-Del attention sequence to escape any running application before a login password can be keyed.

Hardware Solutions : Commodity *vs.* Custom. The safest way to reduce the possibility of password exposure, is to seize it not on a general-purpose computer, but on a dedicated hardware device in the possession of the user.

"Pocket password calculators" have been used for decades by the banking industry for signing high-value electronic transactions, and more recently for generating one-time passwords to gain access to corporate VPNs. Such devices have a small keypad for entering a user PIN, but almost always also contain a custom user-specific private key, which makes them difficult to replace and also sensitive to theft and hardware key recovery attacks.

It is easy to imagine similar keypad-equipped hardware for securely entering one's HPAKE password and for performing all related HPAKE computations, possibly interfacing with a host computer connected to the internet. This would ensure that the password is never exposed, even in case of full compromise of the host computer. A key advantage over earlier "password calculators" is that an HPAKE device would be completely commoditized and contain no user-specific information. User would thus not need to worry about losing the, or having them stolen.

6.2 Real-Time User-Driven Cancellation

Because the HKDF component in HPAKE will not halt spontaneously on all inputs, the client-side UI must include an special button to allow the user to take corrective actions (and optionally, during the registration phase, to make it easier for the user to select the value of τ).

During registration, a "finish" button may serve as a simple and intuitive device for selecting the hardness parameter τ: the user would simply let HKDF.Make run for a while and then click on the finish button, which will cause the system to set τ to the current value of the HKDF loop iterator. The user need not be shown the value of τ, since she has no use for it (except perhaps a vague recollection of what kind of delay she chose, if she suspects she might forget her password).

During authentication, a "cancel" button must be available to let the user stop the process. Since the HKDF.Open function is designed to run forever when called on the wrong inputs, it is up to the user to stop it manually when she realizes that she entered a wrong password. Having a cancel button is always a good idea, since delays can occur for many reasons (*e.g.*, network congestion).

7 Conclusion

The sad reality is that people are not using passwords the way protocol designers and security experts wish they were. It is therefore natural to ask for an authentication protocol that remains as secure as possible under such stringent usage conditions.

Ideally, people should be able to conduct all their online business with a single easy-to-remember password, no matter how numerous or how untrustworthy the web sites they wish to authenticate with.

Just as importantly, the ideal protocol should need zero client-side long-term storage (other than the password), to lessen the security impact in case of loss or theft; this is especially important when traveling. This make a very compelling case for "reusable-password stateless roaming authentication", especially since by far the safest place to keep a password is in one's memory, where there is not much room for more.

Existing password authentication protocols are generally not safe when related passwords are used in multiple contexts. Protocols of the APAKE family come very close, but are still vulnerable to offline dictionary attacks by insiders, unless the password is strong, because they take no measure to limit the rate of such attacks.

Various client-side stop-gap measures have also been proposed, but they invariably have steep additional requirements: for example, browser-based "password managers" require long-term storage on the client side; whereas "anti-phishing" add-ons (intended to save you from mistakenly sending your password to an evildoer on a blacklist) make the tacit assumption that the DNS system and the PKI authorities used in SSL can be trusted. PKI-based solutions

generally require the storage of at least one authentic certificate on the client, too.

Our HPAKE approach is certainly not perfect. However, it has a crucial combination of benefits over the existing alternatives: (1) client and servers have asymmetric secrets; (2) authentication is mutual; (3) no need for any client-side storage; (4) the password is a user secret and can be reused with other servers; (5) outsider attacks can do no better than online password guessing; (6) servers with access to the user registration data can always brute-force the user's password offline, but the presence of a hard function will greatly slow down such attacks; (7) the hard function is user-programmed, giving the user full control over it; (8) the hard function is user-computed, ensuring that it will be applied effectively; (9) the server-side protocol is independent of the hard function, it is lightweight and scales very well.

Our HPAKE protocol is but one example of a possible construction; there are certainly others. Ours has the advantage of being very simple and efficient, but relies heavily (and, in fact, almost exclusively) on the random-oracle model for its security. We leave it as an open question to find other realizations that avoid random oracles but are still reasonably efficient.

We conclude with an obvious but important word of caution: the reusability of weak passwords that HPAKE enables only applies within the confines of HPAKE (and HKDF [9], for local encryption applications). Reusing an HPAKE password on an unsecured web form will void all security guarantees that our cryptography sought to offer.

References

1. Bellare, M., Canetti, R., Krawczyk, H.: A modular approach to the design and analysis of authentication and key exchange protocols. In: Proceedings of the ACM Symposium on the Theory of Computing—STOC 1998. ACM Press, New York (1998)
2. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated key exchange secure against dictionary attacks. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 139–155. Springer, Heidelberg (2000)
3. Bellare, M., Rogaway, P.: Entity authentication and key distribution. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 232–249. Springer, Heidelberg (1994)
4. Bellovin, S.M., Merritt, M.: Encrypted key exchange: Password-based protocols secure against dictionary attacks. In: IEEE Symposium on Security and Privacy—SP 1992, pp. 72–84. IEEE Press, Los Alamitos (1992)
5. Bellovin, S.M., Merritt, M.: Augmented encrypted key exchange. In: ACM Conference on Computer and Communications Security—CCS 1993, pp. 224–250. ACM Press, New York (1993)
6. Bennett, C.H., Brassard, G., Robert, J.-M.: Privacy amplification by public discussion. SIAM Journal of Computing 17(2) (1988)
7. Blake-Wilson, S., Johnson, D., Menezes, A.: Key agreement protocols and their security analysis. In: Darnell, M.J. (ed.) Cryptography and Coding 1997. LNCS, vol. 1355, pp. 30–45. Springer, Heidelberg (1997)

8. Boldyreva, A.: Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In: Desmedt, Y.G. (ed.) PKC 2003. LNCS, vol. 2567, pp. 31–46. Springer, Heidelberg (2003)
9. Boyen, X.: Halting password puzzles. In: USENIX Security Symposium—SECURITY 2007, pp. 119–134. The USENIX Association (2007)
10. Boyen, X.: Hidden credential retrieval from a reusable password. In: ACM Symposium on Information, Computer & Communication Security—ASIACCS 2009. ACM Press, New-York (2009)
11. Boyko, V., MacKenzie, P., Patel, S.: Provably secure password-authenticated key exchange using diffie-hellman. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 156–171. Springer, Heidelberg (2000)
12. Brown, D.R.L.: Prompted user retrieval of secret entropy: The passmaze protocol. Cryptology ePrint Archive, Report 2005/434 (2005), `http://eprint.iacr.org/`
13. Canetti, R., Halevi, S., Katz, J., Lindell, Y., MacKenzie, P.: Universally composable password-based key exchange. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 404–421. Springer, Heidelberg (2005)
14. Canetti, R., Halevi, S., Steiner, M.: Mitigating dictionary attacks on password-protected local storage. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 160–179. Springer, Heidelberg (2006)
15. Canetti, R., Krawczyk, H.: Analysis of key-exchange protocols and their use for building secure channels. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 453–474. Springer, Heidelberg (2001)
16. Chaum, D.: Blind signatures for untraceable payments. In: Advances in Cryptology—CRYPTO 1982, pp. 199–203 (1982)
17. Choo, K.-K.R., Boyd, C., Hitchcock, Y.: Examining indistinguishability-based proof models for key establishment protocols. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 585–604. Springer, Heidelberg (2005)
18. Dean, D., Stubblefield, A.: Using client puzzles to protect TLS. In: USENIX Security Symposium—SECURITY 2001 (2001)
19. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976)
20. Diffie, W., van Oorschot, P., Wiener, M.: Authentication and authenticated key exchanges. Designs, Codes and Cryptography 2, 107–125 (1992)
21. Dwork, C., Naor, M.: Pricing via processing or combating junk mail. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 139–147. Springer, Heidelberg (1993)
22. Ford, W., Kaliski Jr., B.S.: Server-assisted generation of a strong secret from a password. In: Proc. IEEE 9th Int. Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, pp. 176–180. IEEE Computer Society Press, Los Alamitos (2000)
23. Gentry, C., MacKenzie, P., Ramzan, Z.: Password authenticated key exchange using hidden smooth subgroups. In: ACM Conference on Computer and Communications Security—CCS 2005, pp. 299–309. ACM Press, New York (2005)
24. Gentry, C., MacKenzie, P., Ramzan, Z.: A method for making password-based key exchange resilient to server compromise. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 142–159. Springer, Heidelberg (2006)
25. Halevi, S., Krawczyk, H.: Public-key cryptography and password protocols. In: ACM Conference on Computer and Communications Security—CCS 1998, pp. 122–131. ACM Press, New York (1998)
26. Jablon, D.: Strong password-only authenticated key exchange. Computer Communication Review (1996)

27. Jiang, S., Gong, G.: Password based key exchange with mutual authentication. In: Handschuh, H., Hasan, M.A. (eds.) SAC 2004. LNCS, vol. 3357, pp. 267–279. Springer, Heidelberg (2004)
28. Juels, A., Brainard, J.: Client puzzles: A cryptographic defense against connection depletion attacks. In: Proceedings of NDSS 1999, pp. 151–165 (1999)
29. Katz, J., Ostrovsky, R., Yung, M.: Efficient password-authenticated key exchange using human-memorable passwords. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 475–494. Springer, Heidelberg (2001)
30. Krawczyk, H.: HMQV: A high-performance secure Diffie-Hellman protocol. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 546–566. Springer, Heidelberg (2005)
31. LaMacchia, B., Lauter, K., Mityagin, A.: Stronger security of authenticated key exchange. In: Susilo, W., Liu, J.K., Mu, Y. (eds.) ProvSec 2007. LNCS, vol. 4784, pp. 1–16. Springer, Heidelberg (2007)
32. Law, L., Menezes, A., Qu, M., Solinas, J., Vanstone, S.A.: An efficient protocol for authenticated key agreement. Designs, Codes and Cryptography 28(2), 119–134 (2003)
33. MacKenzie, P.: More efficient password-authenticated key exchange. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 361–377. Springer, Heidelberg (2001)
34. MacKenzie, P., Patel, S., Swaminathan, R.: Password-authenticated key exchange based on RSA. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 599–613. Springer, Heidelberg (2000)
35. MacKenzie, P., Shrimpton, T., Jakobsson, M.: Threshold password-authenticated key exchange. Journal of Cryptology 19(1), 27–66 (2006)
36. Maurer, U.: Information-theoretically secure secret-key agreement by not authenticated public discussion. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 209–225. Springer, Heidelberg (1997)
37. Menezes, A.: Another look at HMQV. Cryptology ePrint Archive, Report 2005/205 (2005), `http://eprint.iacr.org/`
38. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1997)
39. Naor, M., Pinkas, B.: Visual authentication and identification. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 322–336. Springer, Heidelberg (1997)
40. Pinkas, B., Sander, T.: Securing passwords against dictionary attacks. In: ACM Conference on Computer and Communications Security—CCS 2002, pp. 161–170. ACM Press, New York (2002)
41. Provos, N., Mazières, D.: A future-adaptable password scheme. In: USENIX Technical Conference—USENIX 1999 (1999)
42. Yee, K.-P., Sitaker, K.: Passpet: Convenient password management and phishing protection. In: Symposium On Usable Privacy and Security—SOUPS 2006. ACM Press, New York (2006)