Asymmetric PAKE with low computation *and* communication

Bruno Freitas Dos Santos[1], Yanqi Gu[1], Stanislaw Jarecki[1], and Hugo Krawczyk[2]

[1] University of California, Irvine. Email: {brunof,yanqig1,sjarecki}@uci.edu.
[2] Algorand Foundation. Email: hugokraw@gmail.com.

Abstract. In Crypto'21 Gu, Jarecki, and Krawczyk [25] showed an *asymmetric* password authenticated key exchange protocol (aPAKE) whose computational cost matches (symmetric) password authenticated key exchange (PAKE) and plain (i.e. unauthenticated) key exchange (KE). However, this minimal-cost aPAKE did not match prior aPAKE's in round complexity, using 4 rounds assuming the client initiates compared to 2 rounds in an aPAKE of Bradley et al. [13].
In this paper we show two aPAKE protocols (but not *strong* aPAKEs like [30, 13]), which achieve optimal computational cost *and* optimal round complexity. Our protocols can be seen as variants of the *Encrypted Key Exchange* (EKE) compiler of Bellovin and Merritt [7], which creates password-authenticated key exchange by password-encrypting messages in a key exchange protocol. Whereas Bellovin and Merritt used this method to construct a PAKE by applying password-encryption to KE messages, we construct an aPAKE by password-encrypting messages of a *unilaterally authenticated* Key Exchange (ua-KE). We present two versions of this compiler. The first uses *salted* password hash and takes 2 rounds if the server initiates. The second uses unsalted password hash and takes a single simultaneous flow, thus *simultaneously matching the minimal computational cost and the minimal round complexity of PAKE and KE.*
We analyze our aPAKE protocols assuming an Ideal Cipher (IC) on a group, and we analyze them as modular constructions from ua-KE realized via a universally composable Authenticated Key Exchange where the server uses *one-time keys* (otk-AKE). We also show that one-pass variants of 3DH and HMQV securely realize otk-AKE in the ROM. Interestingly, the two resulting concrete aPAKE's use the exact same protocol messages as variants of EKE, and the only difference between the symmetric PAKE (EKE) and asymmetric PAKE (our protocols) is in the key derivation equation.

1 Introduction

Password authenticated key exchange (PAKE) lets two parties establish a secure shared session key if and only if they hold the same (possibly low-entropy) password. The *asymmetric* password authenticated key exchange

protocols (aPAKE) is a client-server variant of such protocol where the input to the server party is a one-way function of the password, a.k.a., a *password hash*, and the protocol establishes a shared key iff the client's input is a preimage of the server's input. Both PAKE's and aPAKE's have been extensively studied in the crypto literature, starting from respectively [7] and [29], but recently there has been a renewed interest in aPAKE's due to the weaknesses of current password authentication methods and to the ongoing PAKE standardization effort of the Internet Engineering Task Force [39]. Perhaps the most striking vulnerabilities of the current PKI-based "password-over-TLS" authentication practice, where the client sends its password over a TLS connection to the server, are that it enables phishing attacks against clients who establish a TLS connection with the wrong party, and that it discloses password cleartexts on the server, exposing them to server-side attacks. (To see why the latter might be a problem consider that even security-conscious companies were known to accidentally store large quantities of plaintext passwords [1, 2].)

The recent work of Gu, Jarecki, and Krawczyk [25] considered minimal-cost aPAKE's, and they showed an aPAKE protocol *KHAPE* which nearly matches the computational cost of unauthenticated key exchange (KE), namely Diffie-Hellman (uDH), which is 1fb+1vb exp per party (i.e., 1 fixed-base and 1 variable-base exponentiation). The KE cost is a lower-bound for both PAKE and aPAKE because aPAKE $\Rightarrow$ PAKE $\Rightarrow$ KE. However, the minimal-cost aPAKE protocol of [25] is not close to KE in round complexity. Indeed, the aPAKE of [25] takes 3 rounds assuming the server initiates the protocol, while uDH takes a single simultaneous flow, where each party sends a single protocol message without waiting for the counterparty. Note that this minimal round complexity is achieved by minimal-cost universally composable (UC) PAKE's, including EKE [7, 6, 37], SPAKE2 [4, 3], and TBPEKE [38, 3].[3]

Our Contributions. We show that cost-optimal aPAKE does not have to come at the expense of round complexity. We do so with two new aPAKE constructions, called OKAPE and aEKE, which are generic compilers that construct aPAKE's from any key-hiding *one-time-key* Authenticated Key Exchange (*otkAKE*). Both constructions use the Random Oracle Model (ROM) and an Ideal Cipher (IC) on message spaces formed by otkAKE public keys, and in the case of aEKE also on the space(s) of otkAKE protocol messages. We define the notion of key-hiding otkAKE as a relaxation of the UC key-hiding AKE of [25], and we show that it is realized by "one-pass" variants of 3DH and HMQV which were shown as UC key-hiding AKEs in [25].

The two compilers instantiated with one-pass HMQV produce two concrete aPAKE schemes which we call OKAPE-HMQV and aEKE-HMQV. Both protocols have close to optimal computational cost of 1fb+1vb exp for the client and 1fb+1mvb exp for the server, where mvb stands for multi-exponentiation with two bases. Moreover, protocol aEKE-HMQV needs only a single simultaneous flow

[3] Abdalla et al. [3] show that SPAKE2 [4] and TBPEKE [38] realize a relaxed version of the UC PAKE functionality of Canetti et al. [15].

of communication, hence aEKE-HMQV matches the lowest cost KE and PAKE protocols in both computation and round complexity.

Protocol OKAPE requires 2 communication rounds if the server initiates the protocol, and 3 if the client does. However, protocol aEKE uses *unsalted* password hashes, whereas OKAPE supports *(publicly) salted* password hashes, which have several security and operational benefits over unsalted ones (see Note 1 below). Note that every aPAKE can be generically transformed to support a publicly salted hash if the server first sends the salt to the client and the two parties run aPAKE on the password appended by the salt. However, among prior UC aPAKE's that use unsalted password hashes [24, 32, 28, 41], only the aPAKE of Jutla and Roy [32] and Hwang et al. [28] match the round complexity of OKAPE-HMQV after this transformation, but they do not match its computational cost: The PAKE-to-aPAKE compiler of [28] instantiated with a minimal-cost PAKE has a total computational cost of 3fb+3vb exps, i.e. 50% more than uDH, while the aPAKE of [32] is significantly more expensive, in particular because it uses bilinear maps. This generic transformation can also be applied to aEKE-HMQV, and the resulting protocol would match both the rounds and exponentiation count of OKAPE-HMQV, but OKAPE-HMQV uses only one ideal cipher operation per party whereas aEKE-HMQV uses two, hence the latter is preferable if the cost of IC on a group is not negligible.

The only prior UC aPAKE's that natively support salted hashes with 3 or fewer communication rounds is the 3-round protocol OPAQUE of Jarecki et al. [30, 31] and the 2-round CKEM-based protocol of Bradley et al. [13]. Both of these protocols have at least 2 times higher computational costs than uDH. However, both [30] and [13] provide *strong* aPAKEs (saPAKE), where the salt in the password hash is private, whereas OKAPE supports publicly salted hash and aEKE supports only unsalted hash, see Note 1 below.

In table 1 we compare efficiency and security properties of prior UC aPAKE's and the concrete protocols we propose. Note that all schemes which achieve explicit authentication for only one party can also achieve it for the other using one additional key confirmation flow. Note also that any single-flow aPAKE can be transformed so it achieves explicit authentication for both parties in 3 flows, regardless of which party starts. In the table we do not include aPAKE schemes which were not proven in UC models so far, including VPAKE [8] or PAK-X [12], but both schemes are slightly costlier than e.g. KC-SPAKE2+ [41], see e.g. [13] for exact cost comparisons.

Main Idea: Encrypted Key Exchange paradigm for aPAKE. Our protocols are compilers which build aPAKE's from any key-hiding otkAKE, i.e. an AKE where one party uses a one-time key. In both protocols server S picks a one-time public key pair (b, B) and sends the public key B encrypted under a password hash h to client C, who decrypts it under a hash of its password pw. C also has a long-term private key a derived as a password hash as well, i.e. $(h, a) = \mathsf{H}(pw)$, and S holds the corresponding public key A together with h in the password file for this client. The two parties then run a *key-hiding* otkAKE on respective inputs (a, B) and (b, A), but here the two compilers diverge: In

scheme	client(1)	server(1)	rounds(2)	salting	EA(3)	assump.	model
aEKE-HMQV [*]	1f+1.2v	1f+1.2v	1	none	none	gapDH	RO/IC
Jutla-Roy[32]	O(1)	O(1)	1	none	none	XDH	RO
KC-SPAKE2+[41]	2f+2v	2f+2v	3(C)	none	C+S	CDH	RO
OKAPE-HMQV [*]	1f+1.2v	1f+1.2v	2(S)	public	S	gapDH	RO/IC
Hwang[28] +EKE[7]	2f+1v	1f+2.2v	2(S)	public	S	CDH	RO/IC
KHAPE-HMQV[25]	1f+1.2v	1f+1.2v	3(S)	public	C+S	gapDH	RO/IC
CKEM-saPAKE[13]	10f+1v	2f+2v	2(C)	private	C	sDH,DDH	RO
OPAQUE-HMQV[31]	2f+2.2v	1f+2.2v	3(C)	private	C+S	OM-DH	RO

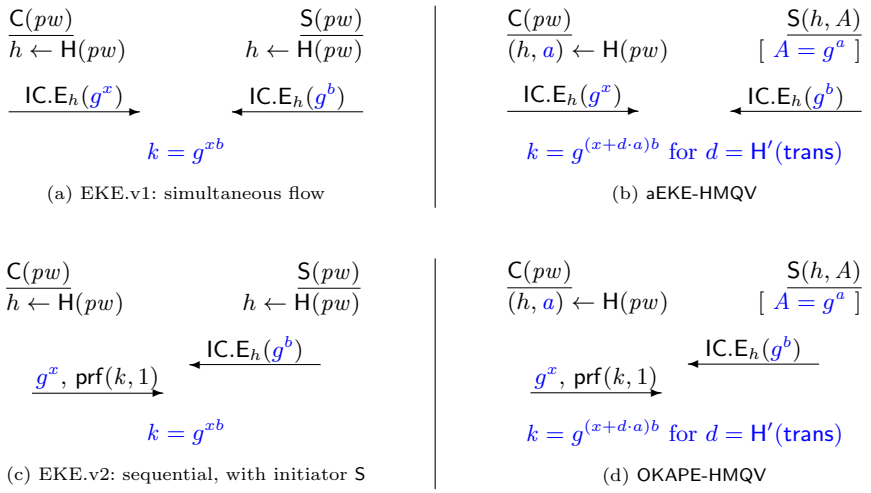
Table 1. Comparison of UC aPAKE schemes, with our schemes marked [*]: (1) f,v denote resp. fixed-base and variable-base exponentiation, two-base multi-exponentiation is counted as 1.2v, O(1) stands for significantly larger costs including bilinear maps; (2) x(C) and x(S) denote x rounds if respectively client starts or server starts, while "1" denotes a single-flow protocol; (3) EA column lists the parties that explicitly authenticate their counterparty at protocol termination. OPAQUE-HMQV appeared in [30], but above we give optimized performances characteristics due to [31].

OKAPE the otkAKE subprotocol is executed in a black-box way, and it is followed by explicit key confirmation message from C to S, whereas in aEKE each otkAKE subprotocol message is encrypted under the password hash by its sender and decrypted under the password hash by its receiver, and no key confirmation message is needed for security. The protocols are shown secure if password-encryption is implemented with an Ideal Cipher on the appropriate message domain, which consists of one-time public keys and/or protocol messages of the underlying otkAKE. Finally, the aEKE compiler requires the key-hiding otkAKE to satisfy a *random transcript* property, i.e. that protocol messages are indistinguishable from uniform over their message spaces.

Note that in both protocols S and C start on resp. inputs A and a and run the following subprotocol: (1) S picks a one-time key pair (b, B) and sends B to C, and (2) the two run otkAKE on resp. (a, B) and (b, A). This subprotocol forms an Authenticated Key Exchange with *unilateral authentication* (ua-KE), where C is authenticated to S but not vice versa. Viewed in this way, protocol aEKE can be seen as an application of the same paradigm as the *Encrypted Key Exchange (EKE)* of Bellovin and Merritt [7]. EKE is a compiler which constructs a (symmetric) PAKE from any random-transcript KE: Each party runs the underlying KE but encrypts protocol messages using a password as a key. This creates a UC PAKE if the encryption is an IC on the KE protocol message space [6, 37]. Protocol aEKE utilizes the exact same methodology of IC-encryption of KE protocol messages with a password (or its hash), but applied to ua-KE instead of KE, and we show that this creates a UC *asymmetric* PAKE.

If an EKE is applied to a single simultaneous flow, i.e. 1-round, KE like uDH, it creates 1-round PAKE. In the same way our aEKE compiler creates 1-round aPAKE given a 1-round ua-KE. On the other hand, if EKE instantiated with

uDH is executed sequentially then the responder party can send its DH message without IC-encrypting it under the password if it attaches a key confirmation message the response [7, 6]. The same trade-off is done by OKAPE compared to aEKE: OKAPE forgoes on IC-encryption of C's ua-KE message but requires C to send a key confirmation message instead.



(a) EKE.v1: simultaneous flow

(b) aEKE-HMQV

(c) EKE.v2: sequential, with initiator S

(d) OKAPE-HMQV

Fig. 1. Symmetric PAKE: *EKE* (a,c) vs. our asymmetric PAKE's (b,d)

These parallels are easy to see if the one-pass HMQV instantiations of aEKE and OKAPE are put side-by-side the two variants of EKE instantiated with uDH as KE, see Figure 1.[4] Since both EKE and our protocols are compilers, resp. from KE and ua-KE, we highlight the underlying uDH instantiation of KE and the one-pass HMQV instantiation of ua-KE in these figures in blue. The choice of variable names g^x and g^b in the Diffie-Hellman key agreement comes from one-pass HMQV, where g^a and g^b are resp. the permanent public key of C and the one-time public key of S, while g^x is the Diffie-Hellman contribution of C. Intuitively, a corresponding g^y contribution of S is not needed because the *ephemeral* key g^b already plays this role.

The security of our aPAKEs holds for essentially the same reasons as the security of EKE: (1) security against passive attackers holds regardless of pw by the passive security of the underlying (ua-)KE; (2) if encryption is an ideal cipher then any ciphertext sent by an attacker to C decrypts to a random group element $B' = g^{b'}$ on all passwords except the one used by an attacker in encryption, so an attack on such sessions would be an attack on a passively observed otkAKE instance; (3) the attacker can encrypt a chosen g^b value under a single password,

[4] Actual protocols diverge from Fig. 1 in some technicalities, e.g. session key derivation uses a hash of k, but crucially H inputs include a *salt* in OKAPE-HMQV and server/user identifiers in aEKE-HMQV: We come back to this last point below.

but in the IC model the simulator can observe this and extract a unique password guess which the attacker tests in such protocol instance; (4) same arguments work regarding attacks on S in aEKE, while in OKAPE the client's key confirmation message commits the attacker to a session key, which implies a single input pair (a, B) for which this session key is correct, which in turn commits to a single password from which (a, B) are derived.

Although our protocols can be seen as applications of EKE compiler to ua-KE, we analyze them as compilers from otkAKE for several reasons: First, otkAKE is a simpler notion which can be realized with a single protocol flow; Second, otkAKE yields ua-KE (see above) while the converse is not clear; Third, setting the boundary around otkAKE lets us treat it as a black box in OKAPE compiler, because S's one time key g^b, which is the only part that OKAPE wraps using IC encryption is an input to otkAKE, and not its protocol message. In aEKE the otkAKE subprotocol is not used as a black-box, because its protocol messages are IC-encrypted, and this compiler is secure only if otkAKE has a random-transcript property. We prove aEKE secure only for otkAKE realized with a single-flow protocol, which includes both our otkAKE instantiations, i.e. one-pass 3DH and HMQV. Although we believe that this compiler works for multi-round protocols as well, we show it only for single-round otkAKE to limit the complexities in the security argument, which arise from the non-black-box use of otkAKE in this compiler.

Similarities to OPAQUE and KHAPE. Our protocols are also closely related to saPAKE protocol OPAQUE [30] and aPAKE protocol KHAPE [25]. Both of these protocols were compilers from AKE (the OPAQUE protocol in addition uses an Oblivious PRF), where passwords are used to encrypt the client's private key a and the server's public key B, the corresponding keys A and b are held in a password file held by S for this client C, and the key establishment comes from AKE run on these inputs. Protocol KHAPE can be seen as a variant of OPAQUE without the Oblivious PRF. In that case security degrades from saPAKE to aPAKE, but the resulting aPAKE can have minimal cost (i.e. $\approx$ KE) if C's AKE inputs (a, B) are delivered from S to C in an envelope, IC-encrypted under the password, and if the AKE protocol is *key-hiding*, i.e. even an active attacker cannot tell what keys $(sk_{\mathsf{P}}, pk_{\mathsf{CP}})$ an attacked party P assumes except if the attacker knows the corresponding pair $(pk_{\mathsf{P}}, sk_{\mathsf{CP}})$. The reason the KHAPE compiler needs the key-hiding property of AKE is to avoid off-line attacks, because if each password decrypts the envelope sent to the client into some pair (a', B'), there must be no way to test which pair corresponds to either the client or the server keys unless via an active attack which tests at most one of these choices.

Our compilers OKAPE and aEKE are *refinements of the KHAPE compiler*: First, instead of permanent envelope in the password file that encrypts (and authenticates) a permanent server public key g^b, we ask the server to create one-time key per each execution, and IC-encrypt it under a password hash stored in the password file. Replacing key-hiding AKE with key-hiding one-time-key AKE reduces complexity because it can be instantiated with a single C-to-S message.

In addition, the IC encryption with subsequent otkAKE together implement implicit S-to-C authentication: If the attacker does not encrypt $B = g^b$ under C's password then C will decrypt it into a random key $B' = g^{b'}$, for which the attacker cannot compute the corresponding session key because it does not know b'. This lets us eliminate the S-to-C key confirmation message in KHAPE and leads to OKAPE. If in addition C's AKE message is IC-encrypted under a password hash then we eliminate also the C-to-S key confirmation message in KHAPE and we get the aEKE protocol, an aPAKE which is non-interactive *and* has optimal computation cost.

Note 1: Salted and unsalted password hashes vs. round complexity. The UC aPAKE model of Gentry et al. [24] does not enforce salting of password hashes, which allows their precomputation and an immediate look-up once the server storage is breached. By contrast, Jarecki et al. [30] proposed a UC *strong* aPAKE model (saPAKE), where each password file includes a random and *private* salt value s, and the password hash involves this salt and cannot be precomputed without it. Our protocols aEKE and OKAPE are just aPAKEs, not saPAKEs, but they can support *public* salting of the password hash, which has security advantages over unsalted hash. Looking more closely, the aPAKE model of [24] enforces that a single real-world offline dictionary attack test corresponds not only to a single password guess pw^* but also a single tuple $(\mathsf{S}, \mathsf{uid})$ where S is an identifier of a server S and uid is a *userID* with which S associates a password file. (This can be seen in command $(\mathsf{OfflineTestPwd}, \mathsf{S}, \mathsf{uid}, pw^*)$ to the aPAKE functionality of [24], included in Fig. 9 in Section A.) This means that a password hash in UC aPAKE, at least as defined by [24], cannot be implemented e.g. simply as $h = \mathsf{H}(pw)$ but in the very least as $h = \mathsf{H}(\mathsf{S}, \mathsf{uid}, pw)$, so that a single H computation corresponds to a single password guess pw and a single account (S, uid). This is indeed how we implement the RO hash in protocol aEKE, see Section 4.

However, such implementation has some negative implications, stemming from the fact that C has to know values $(\mathsf{S}, \mathsf{uid})$ in the protocol. (This is reflected in the aPAKE functionality realized by protocol aEKE, see Appendix A.) Tying such application-layer values in a cryptographic protocol can be problematic. For example, in some applications it might be fine to equate S with e.g. the server's domain name, but it would be then impossible to modify it, since all users would have to reinitialize and recompute their password hashes. An alternative generic implementation is to use (semi) *public salts* as follows: S can associate each uid account with a random salt s, set the password hash as $h = \mathsf{H}(pw|s)$, attach s in the first S-to-C aPAKE message, and the two parties can then run an *unsalted* aPAKE on a modified password $pw' = pw|s$. Since each s is associated with a unique $(\mathsf{S}, \mathsf{uid})$ pair, each H computation still corresponds to a unique $(\mathsf{S}, \mathsf{uid}, pw)$ tuple, but C does not need values $(\mathsf{S}, \mathsf{uid})$ within the aPAKE protocol, and password hashes do not have to change with changes to identifiers S or uid. Moreover, if the aPAKE protocol runs over a TLS connection then an adversary can find s only via an online interaction with S, and it needs to know the user ID string uid for S to retrieve the uid-indexed password file and send s out. Even

better, if clients update the (s, h) values at each login, then value s the adversary compromises for some user will be obsolete after that user authenticates to S.

However, this implementation requires interaction. Since S sends the first message in OKAPE, attaching s to S's message does not influence the round complexity of OKAPE, and this is indeed how we implement password hashes in that protocol, see Section 3. Every unsalted aPAKE can be transformed to *publicly salted* in this way, but for many aPAKEs, including aEKE, this would imply additional communication rounds.

Note 2: Implicit and explicit authentication vs. round complexity. Note that our round-minimal aPAKE protocol aEKE does not have explicit entity authentication, i.e. each party computes a key and the security implies that only a party with proper credentials can compute that key as well, but they do not get a confirmation that their counterparty can compute the same key and thus is indeed the party they meant to establish a connection with. Key confirmation can be added to any KE protocol, but it adds a round of communication. Likewise, our three-round (if C initializes) aPAKE protocol OKAPE has only C-to-S entity authentication, and adding S-to-C entity authentication would make it a four-round protocol. Therefore the round-reduction advantage of OKAPE over protocol KHAPE of [25] will benefit only those applications where C can use the session key without waiting for S's key confirmation message.

Note 3: Current costs of ideal cipher on groups. Just like EKE [7, 6], our protocols rely on an ideal cipher on group elemnets. Implementing an ideal cipher on elliptic curve groups, which are of most interest for current aPAKE proposals, is non-trivial and current techniques for implementing them incur non-negligible costs in computation and sometimes in bandwidth expansion as well. We discuss several implementation options for group IC in Section 6, but to give an example, using the Elligator2 method [9] each IC operation can cost ≈10-15% of 1vb exp and it requires resampling of the encrypted random group element with probability 1/2. Thus we can estimate the total computational cost of OKAPE-HMQV with this IC implementation as (expected) 2fb+1.15vb for S and 1fb+1.15vb for C, and of aEKE-HMQV as 2fb+1.30vb per each party. However, the overhead of IC might be significantly smaller in the case of other settings of interest, like lattice cryptosystems.

Organization. In Section 2, we define key-hiding one-time-key AKE as a UC notion, and we show that 2DH and one-pass HMQV both securely realize this notion under the Gap DH assumption in ROM. In Section 3 and Section 4, we show our two compilers from otkAKE to asymmetric PAKE, namely OKAPE and aEKE. In Section 5 we describe two concrete aPAKE protocol proposals, OKAPE-HMQV and aEKE-HMQV, which instantiate OKAPE and aEKE with one-pass HMQV as the otkAKE. In Section 6 we discuss possible implementation choices of an ideal cipher encryption on elliptic curve groups. Finally, in Appendix A we include our definition(s) of the UC aPAKE model, and in Appendix B we include the overview of the security proof for protocol

OKAPE. For space constraint reasons we defer all security proofs to the full version [23], including an extension which instantiates otkAKE based on SKEME [34], which allows for aPAKE construction based solely on KEM.

2 Key-hiding one-time-key AKE

We define key-hiding *one-time-key* Authenticated Key Exchange (otkAKE), as an asymmetric variant of the universally composable key-hiding AKE defined in [25]. We denote the otkAKE functionality $\mathcal{F}_{\mathsf{otkAKE}}$ and we include it in Figure 2. An AKE functionality allows parties to generate public key pairs (this is modeled by environment query Init to the functionality). These keys can be compromised, modeled by adversarial query Compromise. However, this is the key difference between our (key-hiding) otkAKE functionality and the (key-hiding) AKE functionality of [25], here we distinguish two types of keys, the long-term keys which can be compromised by the adversary, and the ephemeral keys which cannot. We arbitrarily call the first type "client keys" and the second "server keys" because this is how we will use an otkAKE protocol in the context of our otkAKE-to-aPAKE compilers in Section 3 and 4, i.e. clients will use long-term keys and servers will use ephemeral keys in both of these applications of otkAKE.

As in [25], any party P holding a key pair indexed by the public key pk_{P} , whether a long-term one or an ephemeral one, can start a session using such key, and using also some key pk_{CP} as the public key of the counterparty that P expends on this session. This is modeled by the environment's command $(\mathsf{NewSession}, \mathsf{sid}, \mathsf{CP}, \mathsf{role}, pk_{\mathsf{P}}, pk_{\mathsf{CP}})$ to P, where sid is the unique session identifier, CP is the supposed identifier of the counterparty, and role is either `cl` or `sr`, defining if P is supposed to run the long-term-key party or the ephemeral-key party. (As we can see below, the protocols realizing this functionality can be asymmetric, so parties act differently based on that role bit.) As in [25], the functionality marks this session as initially fresh, creates an appropriate session record and picks a random function $R_{\mathsf{P}}^{\mathsf{sid}}$ (whose meaning we will explain shortly). Crucially the functionality only sends $(\mathsf{NewSession}, \mathsf{sid}, \mathsf{P}, \mathsf{CP}, \mathsf{role})$ to the adversary, i.e. the adversary only learns which party P wants to authenticate, which party CP they intend to communicate with, what session identifier sid they use, and whether they play the client and the server role, but the adversary does *not* learn the *keys* this party uses, neither their own key pk_{P} nor the key pk_{CP} this party expects of its counterparty. This, exactly as in [25], models the *key-hiding* property of the AKE's which are required in our AKE-to-aPAKE compiler constructions.

Next, if an adversary actively attacks session $\mathsf{P}^{\mathsf{sid}}$, as opposed to passively observing its interaction with some other session $\mathsf{CP}^{\mathsf{sid}}$, this is modeled by the adversarial query Interfere, and its effect is that session $\mathsf{P}^{\mathsf{sid}}$ is marked as interfered. The consequence of this marking comes in when the session terminates (i.e. if the adversary delivers all messages this party expects) and outputs a key, which is modeled by adversarial query NewKey. Namely, if a

PK stores all public keys created in Init; CPK stores all compromised keys;
$PK^{\mathtt{cl}}_{\mathsf{P}}$ stores P's permanent public keys; $PK^{\mathtt{sr}}_{\mathsf{P}}$ stores P's ephemeral public keys;

Keys: Initialization and Attacks

On $(\mathsf{Init}, \mathsf{role})$ from P:

If $\mathsf{role} \in \{\mathtt{cl}, \mathtt{sr}\}$ send $(\mathsf{Init}, \mathsf{P}, \mathsf{role})$ to $\mathcal{A}$, let $\mathcal{A}$ specify pk s.t. $pk \notin PK$, add pk to PK and $PK^{\mathsf{role}}_{\mathsf{P}}$, and output (Init, pk) to P. If P is corrupt then add pk to CPK.

On $(\mathsf{Compromise}, \mathsf{P}, pk)$ from $\mathcal{A}$: [*this query must be approved by the environment*]

If $pk \in PK^{\mathtt{cl}}_{\mathsf{P}}$ then add pk to CPK.

Login Sessions: Initialization and Attacks

On $(\mathsf{NewSession}, \mathsf{sid}, \mathsf{CP}, \mathsf{role}, pk_{\mathsf{P}}, pk_{\mathsf{CP}})$ from P:

If $pk_{\mathsf{P}} \in PK^{\mathsf{role}}_{\mathsf{P}}$ and there is no prior session record $\langle \mathsf{sid}, \mathsf{P}, \cdot, \cdot, \cdot, \cdot, \cdot \rangle$ then:

- create session record $\langle \mathsf{sid}, \mathsf{P}, \mathsf{CP}, pk_{\mathsf{P}}, pk_{\mathsf{CP}}, \mathsf{role}, \bot \rangle$ marked fresh;
- if $\mathsf{role} = \mathtt{cl}$ and $pk_{\mathsf{CP}} \notin PK^{\mathtt{sr}}_{\mathsf{CP}}$ then re-label this record as $\mathsf{interfered}$;
- initialize random function $R^{\mathsf{sid}}_{\mathsf{P}} : \{0,1\}^3 \to \{0,1\}^\kappa$;
- send $(\mathsf{NewSession}, \mathsf{sid}, \mathsf{P}, \mathsf{CP}, \mathsf{role})$ to $\mathcal{A}$.

On $(\mathsf{Interfere}, \mathsf{sid}, \mathsf{P})$ from $\mathcal{A}$:

If there is session $\langle \mathsf{sid}, \mathsf{P}, \cdot, \cdot, \cdot, \bot \rangle$ marked fresh then change it to $\mathsf{interfered}$.

Login Sessions: Key Establishment

On $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{P}, \alpha)$ from $\mathcal{A}$:

If $\exists$ session record $\mathsf{rec} = \langle \mathsf{sid}, \mathsf{P}, \mathsf{CP}, pk_{\mathsf{P}}, pk_{\mathsf{CP}}, \mathsf{role}, \bot \rangle$ then:

- if rec is marked fresh: If $\exists$ record $\langle \mathsf{sid}, \mathsf{CP}, \mathsf{P}, pk_{\mathsf{CP}}, pk_{\mathsf{P}}, \mathsf{role}', k' \rangle$ marked fresh s.t. $\mathsf{role}' \neq \mathsf{role}$ and $k' \neq \bot$ then set $k \leftarrow k'$, else pick $k \leftarrow_{\mathrm{R}} \{0,1\}^\kappa$;
- if rec is marked $\mathsf{interfered}$ then set $k \leftarrow R^{\mathsf{sid}}_{\mathsf{P}}(pk_{\mathsf{P}}, pk_{\mathsf{CP}}, \alpha)$;
- update rec to $\langle \mathsf{sid}, \mathsf{P}, \mathsf{CP}, pk_{\mathsf{P}}, pk_{\mathsf{CP}}, \mathsf{role}, k \rangle$ and output $(\mathsf{NewKey}, \mathsf{sid}, k)$ to P.

Session-Key Query

On $(\mathsf{SessionKey}, \mathsf{sid}, \mathsf{P}, pk, pk', \alpha)$ from $\mathcal{A}$:

If $\exists$ record $\langle \mathsf{sid}, \mathsf{P}, \cdot, \cdot, \cdot, \cdot, \cdot \rangle$ and $pk' \notin (PK \setminus CPK)$ then send $R^{\mathsf{sid}}_{\mathsf{P}}(pk, pk', \alpha)$ to $\mathcal{A}$.

Fig. 2. $\mathcal{F}_{\mathsf{otkAKE}}$: Functionality for key-hiding one-time key AKE

session is fresh, i.e. it was not actively attacked, then the functionality picks its output session key k as a random string. In other words, this key is secure because there is no interface which allows the adversary to get any information about it. If the adversary passively connects two sessions, e.g. $\mathsf{P}^{\mathsf{sid}}$ and $\mathsf{CP}^{\mathsf{sid}}$, by honestly exchanging their messages, then $\mathcal{F}_{\mathsf{otkAKE}}$ will notice at the NewKey processing that there are two sessions $(\mathsf{P}, \mathsf{sid}, \mathsf{CP}, pk_{\mathsf{P}}, pk_{\mathsf{CP}}, \mathsf{role})$ $(\mathsf{CP}, \mathsf{sid}, \mathsf{P}, pk'_{\mathsf{P}}, pk'_{\mathsf{CP}}, \mathsf{role}')$ that run on matching keys, i.e. $pk_{\mathsf{CP}} = pk'_{\mathsf{P}}$ and $pk'_{\mathsf{CP}} = pk_{\mathsf{P}}$, and complementary roles, i.e. $\mathsf{role} \neq \mathsf{role}'$, then $\mathcal{F}_{\mathsf{otkAKE}}$ sets the key of the session that terminates last as a copy of the one that terminated first. This is indeed as it should be: If two parties run AKE on matching inputs and keys and their messages are delivered without interference they should output the same key.

However, if session $\mathsf{P}^{\mathsf{sid}}$ has been actively attacked, hence it is marked interfered, the session key k output by $\mathsf{P}^{\mathsf{sid}}$ is determined by the random function $R_{\mathsf{P}}^{\mathsf{sid}}$. Specifically, the key will be assigned as the value of $R_{\mathsf{P}}^{\mathsf{sid}}$ on a tuple of three inputs: (1) P's own key pk_{P}, (2) the counterparty's key pk_{CP} which P assumes, and (3) the protocol transcript α which w.l.o.g. is determined by the adversary on this session. This is a non-standard way of modeling KE functionalities, but it suffices for our applications and it allows for inexpensive and communication-minimal implementations as we exhibit with protocols 2DH and one-pass HMQV below. The intuition is that this assures that for any protocol transcript the adversary chooses, each key pair $(pk_{\mathsf{P}}, pk_{\mathsf{CP}})$ which P can use corresponds to an independent session key output of P. Some of these keys can be computed by the adversary via interface SessionKey: The adversary can use it to compute the key P would output on a given transcript α and a given pair $(pk_{\mathsf{P}}, pk_{\mathsf{CP}}) = (pk, pk')$ but only if pk' is either compromised or it is an adversarial key, hence w.l.o.g. we assume the adversary knows the corresponding secret key.

Here is also where our key-hiding one-time-key AKE diverges from the key-hiding AKE notion of [25]: If session $\mathsf{P}^{\mathsf{sid}}$ runs with a client-role then its session key output is guaranteed secure if their assumed counterparty's key pk_{CP} is indeed an ephemeral key of the intended counterparty. Since such keys cannot be compromised, a SessionKey query with $pk' = pk_{\mathsf{CP}}$ will fail the criterion that pk' is compromised or adversarial, hence the adversary has *no interface to learn* P*'s output session key.* However, if the environment (i.e. the higher-level application, like either of our compilers, which utilizes the otkAKE subprotocol) asks $\mathsf{P}^{\mathsf{sid}}$ to run on pk_{CP} which is *not* an ephemeral key of the intended counterparty then $\mathcal{F}_{\mathsf{otkAKE}}$ treats such session as automatically attacked, and marks it interfered. Such session's output key will be computed as $k \leftarrow R_{\mathsf{P}}^{\mathsf{sid}}(pk_{\mathsf{P}}, pk_{\mathsf{CP}}, \alpha)$, and whether or not the adversary can recompute this key via the SessionKey interface depends on whether this (potentially non-ephemeral) key pk_{CP} is compromised or adversarial.

The security of our otkAKE protocols, 2DH and one-pass HMQV, are based on hardness of Gap CDH problem. Recall that Gap CDH is defined as follows: Let g generates a cyclic group $\mathbb{G}$ of prime order p. The Computational Diffie-Hellman

(CDH) assumption on $\mathbb{G}$ states that given $(X, Y) = (g^x, g^y)$ for $(x, y) \leftarrow_{\mathrm{R}} (\mathbb{Z}_p)^2$ it's hard to find $\mathsf{cdh}_g(X, Y) = g^{xy}$. The Gap CDH assumption states that CDH is hard even if adversary has access to a Decisional Diffie-Hellman oracle ddh_g, which on input (A, B, C) returns 1 if $C = \mathsf{cdh}_g(A, B)$ and 0 otherwise.

2.1 2DH as key-hiding one-time-key AKE

We show that key-hiding one-time-key AKE can be instantiated with a "one-pass" variant of the 3DH AKE protocol. 3DH is an implicitly authenticated key exchange used as the basis of the X3DH protocol [36] that underlies the Signal encrypted communication application. 3DH consists of a plain Diffie-Hellman exchange which is authenticated by combining the ephemeral and long-term key of both peers. Specifically, if (a, A) and (b, B) are the long-term key pairs of two communicating parties C and S, and (x, X) and (y, Y) are their ephemeral DH values, then 3DH computes the session key as a hash of the *triple* of Diffie-Hellman values, (g^{xb}, g^{ay}, g^{xy}).

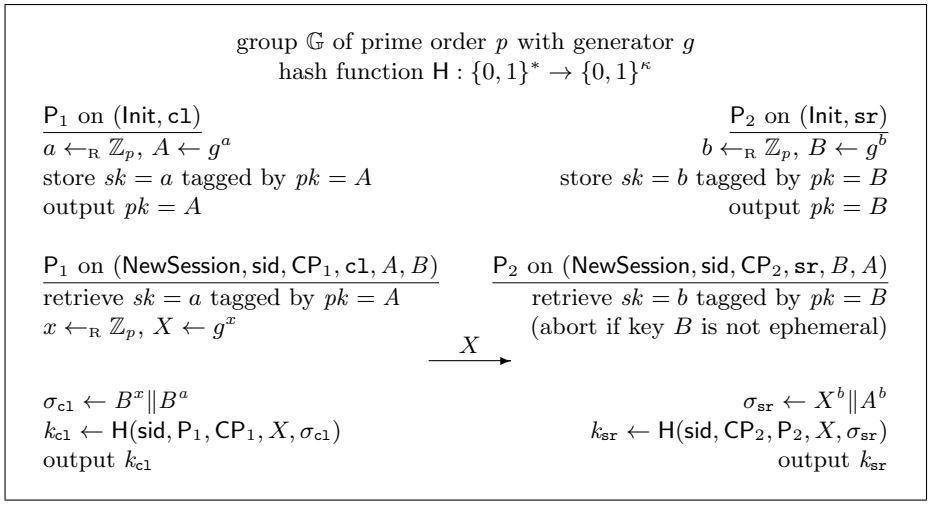


Fig. 3. otkAKE protocol $2DH$

This protocol was shown to realize the key-hiding AKE functionality in [25], and here we show that a one-pass version of this protocol, which we call *2DH*, realizes the key-hiding one-time-key AKE functionality $\mathcal{F}_{\mathsf{otkAKE}}$ defined above. In this modified setting key (b, B) is a one-time key of party S, and hence it can play a double-role as S authenticator *and* its ephemeral DH contribution. Therefore the only additional ephemeral key needed is the (x, X) value provided by C, and 2DH will compute the session key as a (hash of) the *pair* of DH values, (g^{xb}, g^{ab}). See Figure 3 were we describe the 2DH protocol in more detail. In that figure we assume that both C's key (a, A) and S's key (b, B) were created prior to protocol execution, but we note that S's key must be a one-time, i.e.

ephemeral, key, so in practice it should be created just before the protocol starts and erased once the protocol executes.

We capture the security property of 2DH in the following theorem:

Theorem 1. *Protocol 2DH shown in Figure 3 realizes functionality* $\mathcal{F}_{\mathsf{otkAKE}}$*, assuming that the Gap CDH assumption holds on group* $\mathbb{G}$ *and* H *is a random oracle.*

The proof of the above theorem is a close variant of the proof given in [25] that 3DH realizes the key-hiding AKE functionality (where both parties use permanent keys). For the reason of space constraints we include this proof in the full version of the paper [23].

2.2 One-Pass HMQV as key-hiding one-time-key AKE

Similiarly to the case of 3DH, we show that a one-pass version of the HMQV protocol [35, 26] realizes functionality $\mathcal{F}_{\mathsf{otkAKE}}$ under the same Gap CDH assumption in ROM. HMQV is a significantly more efficient AKE protocol compared to 3DH because it replaces 3 variable-base exponentiations with 1 multi-exponentiation with two bases. Just like 3DH, HMQV involves both the ephemeral sessions secrets (x, y) and the long-term keys (a, b), and computes session key using a DH-like formula $g^{(x+da)\cdot(y+eb)}$ where d and e are derived via an RO hash of the ephemeral DH contributions, resp. $X = g^x$ and $Y = g^y$.

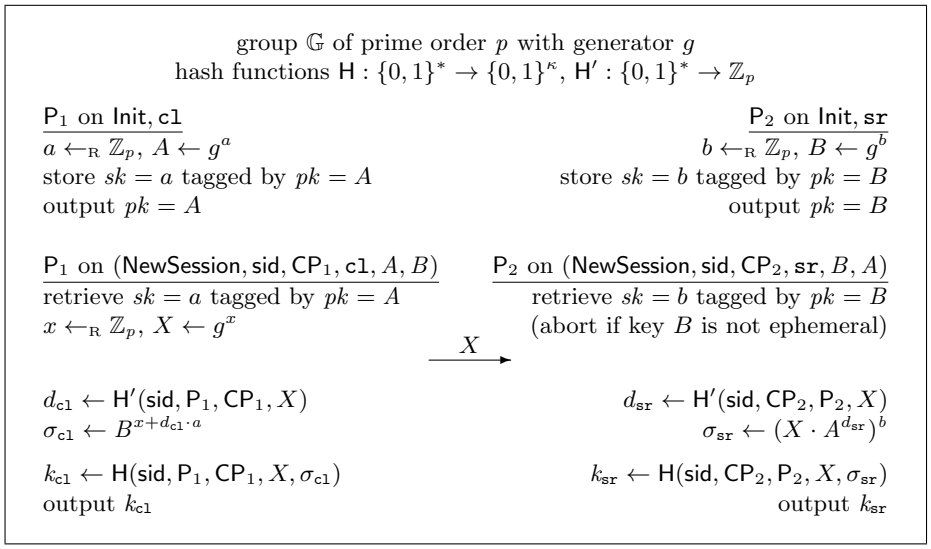


Fig. 4. otkAKE protocol One-Pass HMQV

Gu et al. [25] showed that HMQV realizes the same key-hiding AKE functionality as 3DH, and here we show that a one-pass HMQV realizes the

key-hiding *one-time-key* AKE functionality $\mathcal{F}_{\mathsf{otkAKE}}$. Just like in 2DH, in one-pass HMQV pair (b, B) is a one-time key of party S, which effectively plays the role of both server's public key and its ephemeral DH contribution. Hence just as 2DH, the only ephemeral DH contribution needed is pair (x, X) provided by C, and the session key can be derived as $g^{(x+a)\cdot b}$. The full protocol is shown in Figure 4. As in 2DH we assume that the client and server keys are created before protocol execution, but that the server's key must be a one-time key which is used once and erased afterwards.

We capture the security of one-pass HMQV in the following theorem:

Theorem 2. *Protocol One-Pass HMQV shown in Fig 4 realizes $\mathcal{F}_{\mathsf{otkAKE}}$ if the Gap CDH assumption holds on group $\mathbb{G}$ and H is a random oracle.*

The proof of theorem 2 follows the template of the proof for the corresponding theorem on 2DH security, i.e. Theorem 1. It is also a variant of the similar proof shown in [25] which showed that the full HMQV realizes the permanent-key variant of the key-hiding functionality $\mathcal{F}_{\mathsf{otkAKE}}$ defined therein. Because of space constraints, we defer this proof to the full version of the paper [23].

3 Protocol OKAPE: asymmetric PAKE construction #1

In this section we show how any UC key-hiding one-time-key AKE protocol can be converted into a UC aPAKE, with very small comunication and computational overhead. We call this otkAKE-to-aPAKE compiler OKAPE, which stands for One-time-Key Asymetric PakE, and we present it in Figure 5. As we discussed in the introduction, protocol OKAPE is similar to protocol KHAPE of [25] which is a compiler that creates an aPAKE from any UC key-hiding AKE where both parties use permanent keys. As in KHAPE, the password file which the server S stores and the password which the client C enters into the protocol, allow them to derive AKE inputs (a, B) for C and (b, A) for S, where (a, A) is effectively a client's password-authenticated public key pair and (b, B) is a server's password-authenticated public key pair, and the authenticated key agreement then consists of executing a key-hiding AKE on the above inputs. (The AKE must be key-hiding or otherwise an attacker could link the keys used by either party to a password they used to derive them.)

Protocol otkAKE follows the same general strategy but it differs from KHAPE in (1) how these keys are derived from the client's password and the server's password file, (2) in the type of key-hiding AKE it requires, and (3) whether or not the AKE must be followed by key confirmation messages sent be both parties. In KHAPE the server-side AKE inputs (b, A) were part of the server's password file, and the client-side AKE inputs (a, B) were password-encrypted using an ideal cipher in an envelope $e = \mathsf{IC}.\mathsf{E}_{pw}(a, B)$ stored in the password file and sent from S to C in each protocol instance. Finally, since both public keys were long-term keys, the protocol required each party to send a key-confirmation message and C needed to send its confirmation before S did or otherwise the protocol would be subject to an offline dictionary attack. The first modification made

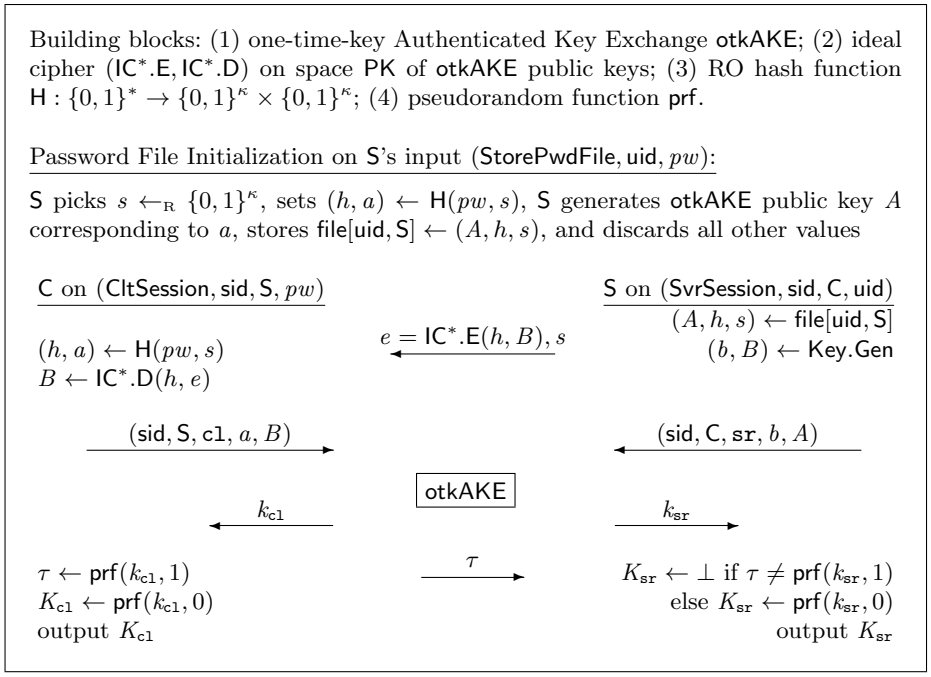


Fig. 5. Protocol OKAPE: Compiler from key-hiding otkAKE to aPAKE

by OKAPE is that the client's private key a is derived directly as a password hash, and does not need to be encrypted in envelope e. Secondly, there is no permanent server's key (b, B). Instead S generates a *one-time* key pair (b, B) at each protocol instance, and authenticates-and-encrypts its public key B under a password by sending to C an envelope $e = \mathsf{IC.E}_h(B)$ where h is a password hash stored in the password file. Since B is now a one-time key, we can replace key-hiding AKE used in KHAPE with a key-hiding one-time-key AKE, which as we saw in Section 2 can be realized with cheaper subprotocols.

More importantly, the IC-encryption of the one-time key B followed by computing the otkAKE session key output by C given input B, implies implicit password-authentication under a unique password: By the properties of the ideal cipher a ciphertext e commits the sender to a single choice of key h (and hence password pw from which h is derived) used to create this ciphertext on a plaintext B chosen by the sender. Hence there can be at most one key h (and thus at most one password pw) s.t. envelope e decrypts to a key B for which the sender knows the corresponding secret key b, and thus can complete the otkAKE protocol ran by C on the key B it decrypts from e. Whereas the protocol still requires a key confirmation by C (otherwise a malicious C could stage an offline dictionary attack once it learned S's session key), the fact that the envelope already implicitly authenticates S implies that it no longer needs a subsequent key confirmation by S.

The main appeal of OKAPE compared to the KHAPE construction in [25] comes from the last implication, i.e. from the fact that we achieve security

without the explicit key confirmation from S. If the OKAPE subprotocol is instantiated with either of the two key-hiding otkAKE protocols of Section 2, the result is a 2-round aPAKE protocol if S is an initiator and a 3-round protocol if C is an initiator (such concrete instantiation is shown in Figure 7 in Section 5). Lastly, because S starts the protocol, protocol OKAPE can use (publicly) *salted* password hash at no extra cost to such instantiations: A random salt value s can be part of the password file, the password hash can be defined as $\mathsf{H}(pw, s)$, and s can be delivered from S to C in S's first protocol message, together with envelope e.

This round-complexity improvement is "purchased" at the cost of two trade-offs. First, in OKAPE server S is only implicitly authenticated to C, and if C requires an explicit authentication of S before C uses its session key then the round reduction no longer applies. Secondly, OKAPE can be slightly more computationally expensive than KHAPE because S needs to generate envelope e on-line, which adds an ideal cipher encryption operation to the protocol cost, and current ideal cipher implementations for e.g. elliptic curve group elements have small but non-negligible costs (see Section 6).

One additional caveat in protocol OKAPE is that because we want C to derive its AKE private key a from a password hash, we must assume that OKAPE generates private keys from uniformly random bitstrings. This is true about any public key generator if that bitstring is treated as the randomness of the key generator algorithm. For some public key cryptosystems, e.g. RSA, this would be a rather impractical representation of the private key, but in the cryptosystems based on Diffie-Hellman in prime-order groups this randomness can be simply equated with the private key.

Theorem 3. *Protocol* OKAPE *realizes the UC aPAKE functionality* $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ *if the AKE protocol realizes functionality* $\mathcal{F}_{\mathsf{otkAKE}}$*, assuming that* prf *is a secure PRF and* IC^* *is an ideal cipher over the space of* otkAKE *ephemeral public keys.*

Functionality $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ is a standard UC aPAKE functionality extended by client-to-server entity authentication. The functionality $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ we use is a modification of the UC aPAKE functionality given by [24], but with some refinements we adopt from [25]. We include this functionality in Appendix A. We provide an abridged version of the proof of Theorem 3 in Appendix B. It describes our simulation strategy and contains the formal definition of our two-part simulator. For a full version, including the intermediary games and (negligible) bounds between the real and ideal-world interaction, we refer the reader to the full proof in the Appendix of [23].

4 Protocol aEKE: asymmetric PAKE construction #2

In Figure 6 we present an asymmetric PAKE protocol that we name *asymmetric encrypted key exchange* (aEKE). It is a close variant of the otkAKE-to-aPAKE compiler OKAPE described in Figure 5 in Section 3. The password file stored by the server also contains client's public key A and a password hash h (which are

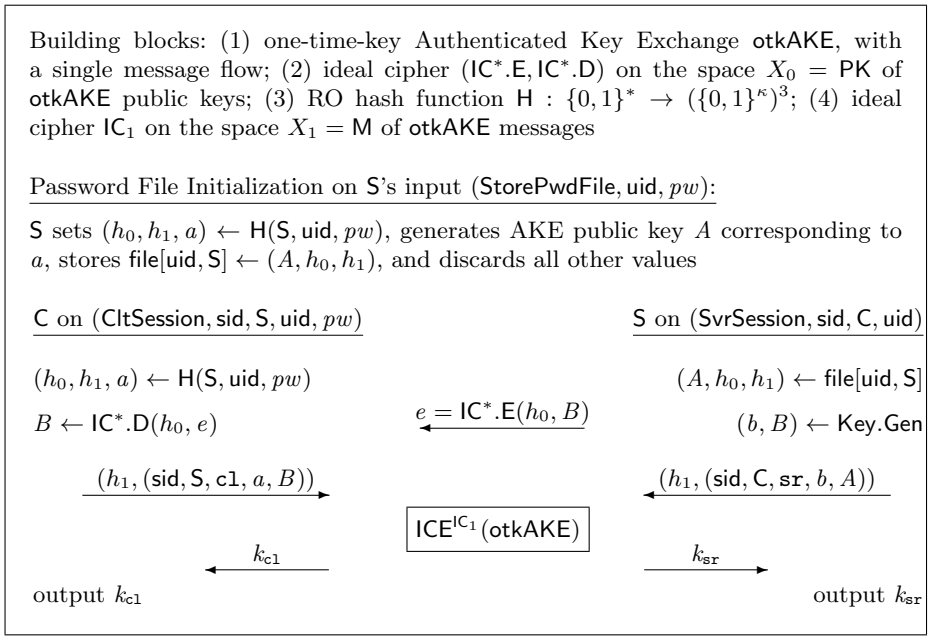


Fig. 6. Protocol aEKE: EKE-style compiler from key-hiding otkAKE to aPAKE

now split into two values h_0, h_1), the server again picks a one-time-key B and sends it over to the client IC-encrypted under the partial password hash h_0, the client derives its private key a via a password hash as well, and the client and server perform otkAKE on the same respective inputs (a, B) and (b, A).

However, the two compilers differ in three important aspects: First, subprotocol otkAKE is not executed as a black box, but it is "IC-encrypted" using the second part h_1 of the password hash as the key. We describe what IC-encrypted protocol is more formally below, but intuitively an *IC-encrypted protocol* Π means that the two parties execute the protocol Π but they use the ideal cipher to encrypt each outgoing message and decrypt each incoming message. The second difference is that the protocol no longer needs C-to-S key confirmation for its security, as was required by protocol OKAPE, which allows for further reduction of round complexity in concrete instantiations.

Lastly, we eliminate the salt value s from the input of the password hash. This last change is done chiefly so that our concrete instantiation of this compiler produces a protocol which requires only a single **simultaneous** flow between the two parties. This will be true as long as the otkAKE subprotocol involves only one message, from C to S, and moreover this message can be generated prior to client learning input B which C derives from the envelope received from S. Note that this is true for our two otkAKE instantiations, namely one-pass HMQV and 2DH. Note that the protocol can be salted in a generic way, i.e. by S storing a salt and sending it to C before the protocol starts, but this would add a S-to-C round of interaction to the protocol (in particular C will need the salt value to

generate h_1 and encrypt its protocol message under it), at which point aEKE would loose its round-complexity advantage over OKAPE.

In Figure 6, and in the security analysis of protocol aEKE, we assume a restricted case of a *single-flow* (C-to-S) realization of subprotocol otkAKE. We believe that the compiler works for multi-round otkAKE subprotocols as well, but dropping this restriction would make the security argument singificantly more complex since our security argument cannot treat the otkAKE subprotocol entirely as a black box, and must explicitly process ideal cipher encryption and decryption of each message in the underlying otkAKE subprotocol.

We stress that, as described in the introduction, even though we drop salting in aEKE our functionality requires that offline password tests correspond to a unique choice of pair $(\mathsf{S}, \mathsf{uid})$, and we enforce this by setting the password hash as $\mathsf{H}(\mathsf{S}, \mathsf{uid}, pw)$.

The *Ideal-Cipher-Encrypted* protocol compiler. The *IC-encrypted* protocol compiler, denoted ICE, takes an ideal cipher IC and any two-party protocol Π and creates a new protocol $\Pi' = \mathsf{ICE}^{\mathsf{IC}}(\Pi)$, which proceeds by running the original protocol Π and encrypting each of its outgoing messages using the ideal cipher IC and decrypting any incoming messages using the same cipher. More specifically, the input of P to protocol Π' is $x' = (k, x)$ where k is a key of an ideal cipher IC and x is P's input in protocol Π. P then runs protocol Π on x, and whenever Π creates an outgoing message $\mathsf{msg}_{\mathsf{out}}$ then P encrypts it as $\mathsf{ciph}_{\mathsf{out}} \leftarrow \mathsf{IC.E}(k, \mathsf{msg}_{\mathsf{out}})$ and sends out ciphertext $\mathsf{ciph}_{\mathsf{out}}$ instead of the original message $\mathsf{msg}_{\mathsf{out}}$. Because P's counterparty is assumed to follow the same protocol, party P parses its incoming message as a ciphertext $\mathsf{ciph}_{\mathsf{in}}$, decrypts it as $\mathsf{msg}_{\mathsf{in}} \leftarrow \mathsf{IC.D}(k, \mathsf{ciph}_{\mathsf{in}})$, and passes $\mathsf{msg}_{\mathsf{in}}$ as an incoming message to protocol Π. Whenever Π terminates with some output this is also the output of protocol Π'.

Observe that the ICE compiler generalizes the Encrypted Key Exchange (EKE) construction of Bellovin and Meritt [7]. The EKE protocol can be seen as protocol $\mathsf{EKE} = \mathsf{ICE}^{\mathsf{IC}}(\mathsf{KE})$, where KE is an unauthenticated Key Exchange, and the password (or its hash) is used as the ideal cipher key. Protocol aEKE is created using exactly the same compiler but applied to otkAKE instead of KE, and it results in asymmetric PAKE instead of symmetric PAKE.

Below we define random-transcript property for single-flow protocol Π. Clearly both 2DH and one-pass HMQV satisfy this property.

Definition 1. ***[random transcript single-flow protocol]*** *Let* M *be the message space of a single-flow protocol Π. We say that protocol Π has a* random transcript *property if for any input x of Π, the message m which Π generates on x is indistinguishable from a message uniformly sampled from* M*, i.e. for any PPT adversary $\mathcal{A}$ and any x, there is a negligible function* negl *such that:*

$$|\Pr[\mathcal{A}(m_0 \leftarrow \Pi(x)) = 1] - \Pr[\mathcal{A}(m_1 \leftarrow_R \mathsf{M}) = 1]| \leq \mathsf{negl}(\kappa)$$

Theorem 4. *Protocol* aEKE *realizes the UC aPAKE functionality $\mathcal{F}_{\mathsf{aPAKE}}$ if the* otkAKE *protocol realizes functionality $\mathcal{F}_{\mathsf{otkAKE}}$ and (1)* otkAKE *protocol uses a*

single client-to-server message, (2) otkAKE *protocol has the random-transcript property, (3)* IC^* *is an ideal cipher over message space of* otkAKE *public keys, (4)* IC_1 *is an ideal cipher over message space* M *of the single-flow* otkAKE.

Because of space constraints the proof of Theorem 4 is deferred to the full version of the paper [23].

5 Concrete aPAKE protocol instantiations

We include two concrete aPAKE protocols we call OKAPE-HMQV and aEKE-HMQV. Protocol OKAPE-HMQV, shown in Figure 7, is an instantiation of protocol OKAPE from Section 3 with one-pass-HMQV as the key-hiding otkAKE (shown in Section 2.2). Protocol aEKE-HMQV, shown in Figure 8, is an instantiation of aEKE from Section 4 with the same one-pass-HMQV. Both of these protocols were shown in a simplified form in Figure 1 in the introduction, but here we show both protocols with all details.

Building blocks: (1) group $\mathbb{G}$ of prime order p with generator g; (2) ideal cipher $(\mathsf{IC}^*.\mathsf{E}, \mathsf{IC}^*.\mathsf{D})$ on space of $\mathbb{G}$; (3) RO hash functions $\mathsf{H}, \mathsf{H}', \mathsf{H}''$ with ranges resp. $\{0,1\}^\kappa \times \mathbb{Z}_p$, $\mathbb{Z}_p$, and $\{0,1\}^\kappa$; (4) pseudorandom function prf.

Password File Initialization on S's input $(\mathsf{StorePwdFile}, \mathsf{uid}, pw)$:

S picks $s \leftarrow_{\mathrm{R}} \{0,1\}^\kappa$, sets $(h, a) \leftarrow \mathsf{H}(pw, s)$, S generates AKE public key $A \leftarrow g^a$, stores $\mathsf{file}[\mathsf{uid}, \mathsf{S}] \leftarrow (A, h, s)$, and discards all other values

C on $(\mathsf{CltSession}, \mathsf{sid}, \mathsf{S}, pw)$		S on $(\mathsf{SvrSession}, \mathsf{sid}, \mathsf{C}, \mathsf{uid})$
$x \leftarrow_{\mathrm{R}} \mathbb{Z}_p$, $X \leftarrow g^x$		$(A, h, s) \leftarrow \mathsf{file}[\mathsf{uid}, \mathsf{S}]$
$(h, a) \leftarrow \mathsf{H}(pw, s)$	$\xleftarrow{e = \mathsf{IC}^*.\mathsf{E}(h, B),\ s}$	$b \leftarrow_{\mathrm{R}} \mathbb{Z}_p$, $B \leftarrow g^b$
$B \leftarrow \mathsf{IC}^*.\mathsf{D}(h, e)$		
$d_{\mathsf{C}} \leftarrow \mathsf{H}'(\mathsf{sid}, \mathsf{C}, \mathsf{S}, X)$	$\xrightarrow{X}$	$d_{\mathsf{S}} \leftarrow \mathsf{H}'(\mathsf{sid}, \mathsf{C}, \mathsf{S}, X)$
$\sigma_{\mathsf{C}} \leftarrow B^{x + d_{\mathsf{C}} \cdot a}$		$\sigma_{\mathsf{S}} \leftarrow (X \cdot A^{d_{\mathsf{S}}})^b$
$k_{\mathtt{cl}} \leftarrow \mathsf{H}''(\mathsf{sid}, \mathsf{C}, \mathsf{S}, X, \sigma_{\mathsf{C}})$		$k_{\mathtt{sr}} \leftarrow \mathsf{H}''(\mathsf{sid}, \mathsf{C}, \mathsf{S}, X, \sigma_{\mathsf{S}})$
$\tau \leftarrow \mathsf{prf}(k_{\mathtt{cl}}, 1)$	$\xrightarrow{\tau}$	$K_{\mathtt{sr}} \leftarrow \perp$ if $\tau \neq \mathsf{prf}(k_{\mathtt{sr}}, 1)$
$K_{\mathtt{cl}} \leftarrow \mathsf{prf}(k_{\mathtt{cl}}, 0)$		else $K_{\mathtt{sr}} \leftarrow \mathsf{prf}(k_{\mathtt{sr}}, 0)$
output $K_{\mathtt{cl}}$		output $K_{\mathtt{sr}}$

Fig. 7. OKAPE with one-pass-HMQV: concrete aPAKE protocol OKAPE-HMQV

Protocol OKAPE-HMQV has 2 flows if the server initiates (and 3 if the client does), while protocol aEKE-HMQV is non-interactive, i.e. each party can send its message without waiting for the counterparty. Note that in both protocols each party uses only 1 fixed-base exponentiation plus 1 variable-base

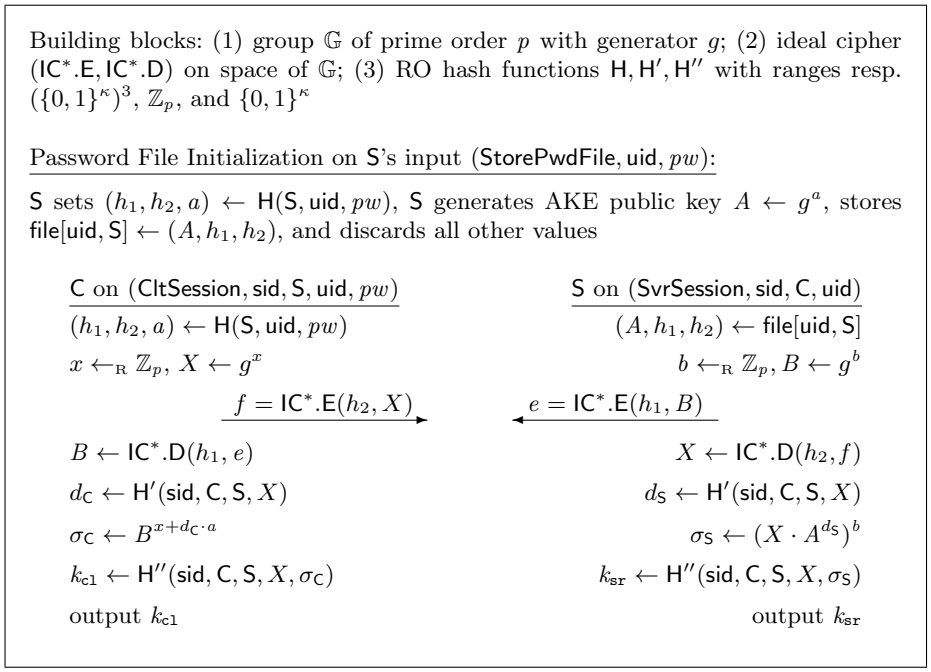


Fig. 8. aEKE with one-pass-HMQV: concrete aPAKE protocol aEKE-HMQV

(multi)exponentiation. In OKAPE-HMQV each party performs one ideal cipher operation: S performs encryption and C decryption, while in protocol aEKE-HMQV each party performs 1 encryption and 1 decryption.

The communication costs are as in Diffie-Hellman key exchange, with one-sided key confirmation and a κ-bit salt value in the case of protocol OKAPE-HMQV. (Recall that OKAPE is a *salted* aPAKE while aEKE is an *unsalted* aPAKE.) Depending on the implementation of an Ideal Cipher encryption on group $\mathbb{G}$, the ciphertext e encrypting B, and in the case of aEKE-HMQV also ciphertext f encrypting X, can introduce additional bandwidth overhead of $\Omega(\kappa)$ bits, and they may also impose non-trivial computational costs on operations $\mathsf{IC}^*.\mathsf{E}$ and $\mathsf{IC}^*.\mathsf{D}$ as well, see Sec. 6.

6 Curve Encodings and Ideal Cipher

Quasi bijections. Protocols OKAPE and aEKE use an Ideal Cipher (IC) on values related to a key-hiding otkAKE subprotocol with which these compiler constructions are instantiated. These values are the server's otkAKE one-time public key B, and in the case of aEKE these are also otkAKE protocol messages. However, since in Section 4 we restrict our claims about aEKE only to the case when subprotocol otkAKE is a single-flow protocol, the IC encryption will be applied only to the client's single otkAKE message. In both instantiations of

otkAKE we exhibit, i.e. 2DH and one-pass HMQV shown in Sections 2.1 and 2.2, the server's public key B is a group element and so is the client' single protocol message X. Hence we need the ideal cipher on a message space which is group $\mathbb{G}$ used in these otkAKE instantiations.

We use the same methodology for implementing an ideal cipher on a group as in [25]. We briefly summarize it here and we refer for more details to [25]. We assume that we have an ideal cipher $\mathsf{IC} = (\mathsf{IC.E}, \mathsf{IC.D})$ which works over fixed-length bitstrings, i.e. space $\{0,1\}^n$ for some n.[5] Then, an ideal cipher on $\mathbb{G}$ can be implemented by encoding plaintext $m \in \mathbb{G}$ as a bitstring of length n, and then apply the ideal cipher IC to the resulting bitstring. The encoding $\mathsf{map} : \mathbb{G} \to \{0,1\}^n$ must be *injective*, i.e. 1-1, so that there exists an (efficient) inverse map $\mathsf{map}^{-1} : \{0,1\}^n \to \mathbb{G}$. The encoding must also be *surjective* (or close) so that every bitstring decodes into a group element, so that e.g. if e is an encryption of $g \in \mathbb{G}$ under key k, the decryption of e under key $k' \neq k$ returns another element in $\mathbb{G}$. If $\mathbb{G}$ is an elliptic curve then we only know examples of *randomized* encodings which satisfy these properties. Formally we define a randomized encoding which is close to a bijection as in [25]:

Definition 2. *[25] A* randomized ε-quasi bijection map *with domain A, randomness space $R = \{0,1\}^\rho$ and range B consists of two efficient algorithms $\mathsf{map} : A \times R \to B$ and $\mathsf{map}^{-1} : B \to A$ with the following properties:*

1. map^{-1} *is deterministic and for all $a \in A, r \in R, \mathsf{map}^{-1}(\mathsf{map}(a,r)) = a$;*
2. map *maps the uniform distribution on $A \times R$ to a distribution on B that is (statistically) ε-close to uniform.*

We say that map is a *quasi bijection* without specifying ε when it is an ε-quasi bijection for negligible ε. Given such encoding a (randomized) ideal cipher $\mathsf{IC}^* = (\mathsf{IC}^*.\mathsf{E}, \mathsf{IC}^*.\mathsf{D})$ on $\mathbb{G}$ can be implemented as $\mathsf{IC}^*.\mathsf{E}(k,m) = \mathsf{IC.E}(k, \mathsf{map}(m;r))$ for random r and $\mathsf{IC}^*.\mathsf{D}(k,c) = \mathsf{map}^{-1}(\mathsf{IC.D}(k,c))$. However, rather than define a new notion of randomized ideal cipher, in protocols OKAPE and aEKE we assume that the ideal cipher on $\mathbb{G}$ is implemented using the above construction IC^* and we argue directly based on the properties of quasi-bijective encoding map and the bitstring ideal cipher IC.

Elliptic curve encodings. There are many well-studied quasi-bijective encodings for elliptic curves in the literature (cf. [40, 14, 22, 9, 42]). We briefly introduce two representative examples and refer to [25] for more details. The **Elligator-squared** method [42, 33] applies to most elliptic curves and implements quasi bijection for the whole group $\mathbb{G}$ of prime order q. It encodes curve points $m \in \mathbb{G}$ as pair of field elements $(u,v) \in \mathbb{Z}_q^2$ using a deterministic function $f : \mathbb{Z}_q \to \mathbb{G}$ s.t. $\mathsf{map}^{-1}(u,v) = m$ iff $m = f(u) + f(v)$. Since u, v are field elements, a further quasi bijection is needed to represent such pair as a bitstring unless q is close to a power of 2. The performance of map^{-1} used in

[5] For $n = 128$ one can assume that e.g. AES is an ideal cipher, while for larger values one has to use domain-extension techniques, e.g. [16, 20, 27, 17] or direct constructions, e.g. [19, 5, 21, 11, 10, 18].

$\mathsf{IC}^*.\mathsf{D}$ depends on function f whose cost is typically dominated by 1 base-field exponentiation, which costs ≈10-15% of a scalar multiplication (a.k.a. a "variable-base exponentiation" in $\mathbb{G}$). The randomized map map used in $\mathsf{IC}^*.\mathsf{E}$ can cost e.g. 3 base-field exponentiations on some curves [42]. The **Elligator2** method [9] is more restrictive, and defines an injective mapping from *half* of the domain $\mathbb{G}$ to integer range $[0, (q-1)/2]$. The advantages of Ellligator2 is that it uses a single field element to represent a group element (thus reducing bandwidth), and that both directions of the map are very efficient, each costing about 1 base-field exponentiation. The disadvantage is that message $m \in \mathbb{G}$ (i.e. S's message B or C's message X) has to be resampled until it lies in the Elligator2 domain.[6] Finally, in a recent work of McQuoid et al. [37] show a "one-time" variant of a randomized ideal cipher on a group, called *Programmable Once Public Function* (**POPF**) therein, which utilizes a single RO-indistinguishable hash onto group $\mathbb{G}$ in both encryption and decryption directions, and as McQuoid et al. show suffices as a replacement for an ideal cipher in the proof that EKE realizes UC PAKE functionality [37]. Because of the similarities between our aPAKE's and EKE, the same POPF notion could suffice in the context of our aPAKEs as well, leading to another method for instantiating the group ideal cipher IC^*, but we leave formal verification that this is the case to future work.

References

1. Facebook stored hundreds of millions of passwords in plain text, `https://www.theverge.com/2019/3/21/18275837/facebook-plain-text-password-storage-hundreds-millions-users`.
2. Google stored some passwords in plain text for fourteen years, `https://www.theverge.com/2019/5/21/18634842/google-passwords-plain-text-g-suite-fourteen-years`.
3. M. Abdalla, M. Barbosa, T. Bradley, S. Jarecki, J. Katz, and J. Xu. Universally composable relaxed password authenticated key exchange. In *Advances in Cryptology - CRYPTO 2020*, pages 278–307, 2020.
4. M. Abdalla and D. Pointcheval. Simple password-based encrypted key exchange protocols. In *Topics in Cryptology – CT-RSA 2005*, pages 191–208. Springer, 2005.
5. E. Andreeva, A. Bogdanov, Y. Dodis, B. Mennink, and J. P. Steinberger. On the indifferentiability of key-alternating ciphers. In *Advances in Cryptology – CRYPTO 2013*, pages 531–550, 2013.
6. M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *Advances in Cryptology – EUROCRYPT 2000*, pages 139–155. Springer, 2000.
7. S. M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *IEEE Computer Society Symposium on Research in Security and Privacy – S&P 1992*, pages 72–84. IEEE, 1992.

[6] We should note that a cost-saving implementation which walks through *consecutive* values e.g. $X_i = g^{x+i}$ for $i = 0, 1, ...$ to find X_i which is the Elligator2 domain, and encrypts that X_i under a password, would leak information about the password if the adversary learns the length of this walk from timing information.

8. F. Benhamouda and D. Pointcheval. Verifier-based password-authenticated key exchange: New models and constructions. *IACR Cryptology ePrint Archive*, 2013:833, 2013.
9. D. J. Bernstein, M. Hamburg, A. Krasnova, and T. Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In *ACM Conference on Computer and Communications Security – CCS 2013*, 2013.
10. D. J. Bernstein, S. Kölbl, S. Lucks, P. M. C. Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F.-X. Standaert, Y. Todo, and B. Viguier. Gimli: a cross-platform permutation. Cryptology ePrint Archive, Report 2017/630, 2017. `http://eprint.iacr.org/2017/630`.
11. G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Keccak. In *Advances in Cryptology – EUROCRYPT 2013*, pages 313–314, 2013.
12. V. Boyko, P. MacKenzie, and S. Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In *Advances in Cryptology – EUROCRYPT 2000*, pages 156–171. Springer, 2000.
13. T. Bradley, S. Jarecki, and J. Xu. Strong asymmetric PAKE based on trapdoor CKEM. In *Advances in Cryptology - CRYPTO 2019*, pages 798–825, 2019.
14. E. Brier, J.-S. Coron, T. Icart, D. Madore, H. Randriam, and M. Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. In *Advances in Cryptology – CRYPTO 2010*, 2010.
15. R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. MacKenzie. Universally composable password-based key exchange. In *Advances in Cryptology – EUROCRYPT 2005*, pages 404–421. Springer, 2005.
16. J.-S. Coron, Y. Dodis, A. Mandal, and Y. Seurin. A domain extender for the ideal cipher. In *Theory of Cryptography Conference – TCC 2010*, pages 273–289, 2010.
17. D. Dachman-Soled, J. Katz, and A. Thiruvengadam. 10-round Feistel is indifferentiable from an ideal cipher. In *Advances in Cryptology – EUROCRYPT 2016*, pages 649–678, 2016.
18. J. Daemen, S. Hoffert, G. V. Assche, and R. V. Keer. The design of Xoodoo and Xoofff. 2018:1–38, 2018.
19. Y. Dai, Y. Seurin, J. P. Steinberger, and A. Thiruvengadam. Indifferentiability of iterated Even-Mansour ciphers with non-idealized key-schedules: Five rounds are necessary and sufficient. In *Advances in Cryptology – CRYPTO 2017*, 2017.
20. Y. Dai and J. P. Steinberger. Indifferentiability of 8-round Feistel networks. In *Advances in Cryptology – CRYPTO 2016*, pages 95–120, 2016.
21. Y. Dodis, M. Stam, J. P. Steinberger, and T. Liu. Indifferentiability of confusion-diffusion networks. In *Advances in Cryptology – EUROCRYPT 2016*, pages 679–704, 2016.
22. P.-A. Fouque, A. Joux, and M. Tibouchi. Injective encodings to elliptic curves. In *Australasia Conference on Information Security and Privacy – ACISP 2013*, 2013.
23. B. Freitas Dos Santos, Y. Gu, S. Jarecki, and H. Krawczyk. Asymmetric PAKE with low computation *and* communication. *IACR Cryptology ePrint Archive*, 2022, 2022. `https://ia.cr/2022`.
24. C. Gentry, P. MacKenzie, and Z. Ramzan. A method for making password-based key exchange resilient to server compromise. In *Advances in Cryptology – CRYPTO 2006*, pages 142–159. Springer, 2006.
25. Y. Gu, S. Jarecki, and H. Krawczyk. KHAPE: Asymmetric PAKE from key-hiding key exchange. In *Advances in Cryptology - Crypto 2021*, pages 701–730, 2021. `https://ia.cr/2021/873`.
26. S. Halevi and H. Krawczyk. One-pass hmqv and asymmetric key-wrapping. In *Advances in Cryptology – PKC 2011*, pages 317–334. Springer, 2011.

27. T. Holenstein, R. Künzler, and S. Tessaro. The equivalence of the random oracle model and the ideal cipher model, revisited. In *STOC 2011*, 2011.
28. J. Y. Hwang, S. Jarecki, T. Kwon, J. Lee, J. S. Shin, and J. Xu. Round-reduced modular construction of asymmetric password-authenticated key exchange. In *Security and Cryptography for Networks – SCN 2018*, pages 485–504. Springer, 2018.
29. D. P. Jablon. Extended password key exchange protocols immune to dictionary attacks. In *6th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 1997)*, pages 248–255, Cambridge, MA, USA, June 18–20, 1997. IEEE Computer Society.
30. S. Jarecki, H. Krawczyk, and J. Xu. OPAQUE: an asymmetric PAKE protocol secure against pre-computation attacks. In *Advances in Cryptology - EUROCRYPT 2018*, pages 456–486, 2018. IACR ePrint version at http://eprint.iacr.org/2018/163.
31. S. Jarecki, H. Krawczyk, and J. Xu. On the (in)security of the diffie-hellman oblivious PRF with multiplicative blinding. In *Public-Key Cryptography - PKC 2021*, pages 380–409, 2021. `https://ia.cr/2021/273`.
32. C. S. Jutla and A. Roy. Smooth NIZK arguments. In *Theory of Cryptography – TCC 2018*, pages 235–262. Springer, 2018.
33. T. Kim and M. Tibouchi. Invalid curve attacks in a GLS setting. In *International Workshop on Security (IWSEC 2015)*, 2015.
34. H. Krawczyk. SKEME: A versatile secure key exchange mechanism for internet. In *1996 Internet Society Symposium on Network and Distributed System Security (NDSS)*, pages 114–127, 1996.
35. H. Krawczyk. HMQV: A high-performance secure Diffie-Hellman protocol (extended abstract). In *Advances in Cryptology – CRYPTO 2005*, pages 546–566. Springer, 2005.
36. M. Marlinspike and T. Perrin. The X3DH key agreement protocol, `https://signal.org/docs/specifications/x3dh/`, 2016.
37. I. McQuoid, M. Rosulek, and L. Roy. Minimal symmetric PAKE and 1-out-of-n OT from programmable-once public functions. In J. Ligatti, X. Ou, J. Katz, and G. Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. `https://eprint.iacr.org/2020/1043`.
38. D. Pointcheval and G. Wang. VTBPEKE: Verifier-based two-basis password exponential key exchange. In *ASIACCS 17*, pages 301–312. ACM Press, 2017.
39. J. Schmidt. Requirements for password-authenticated key agreement (PAKE) schemes, `https://tools.ietf.org/html/rfc8125`, Apr. 2017.
40. A. Shallue and C. van de Woestijne. Construction of rational points on elliptic curves over finite fields. In *ANTS*, 2006.
41. V. Shoup. Security analysis of SPAKE2+. *IACR Cryptol. ePrint Arch.*, 2020:313, 2020.
42. M. Tibouchi. Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings. In *Financial Cryptography – TCC 2014*, pages 139–156, 2014.

A Universally Composable asymmetric PAKE model

We include for reference the UC aPAKE definition in the form of a functionality $\mathcal{F}_{\mathsf{aPAKE}}$, shown in Figure 9. This functionality is largely as it was originally

Password Registration

– On $(\mathsf{StorePwdFile}, \mathsf{uid}, pw)$ from S create record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ marked fresh.

Stealing Password Data [*these queries must be approved by the environment*]

– On $(\mathsf{StealPwdFile}, \mathsf{S}, \mathsf{uid})$ from $\mathcal{A}$, if there is no record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$, return "no password file". Otherwise mark this record $\mathsf{compromised}$, and if there is a record $\langle \mathsf{offline}, \mathsf{S}, \mathsf{uid}, pw \rangle$ then send pw to $\mathcal{A}$.

– On $(\mathsf{OfflineTestPwd}, \mathsf{S}, \mathsf{uid}, pw^*)$ from $\mathcal{A}$, then do:

- If $\exists$ record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ marked $\mathsf{compromised}$, do the following: If $pw^* = pw$ then return "correct guess" to $\mathcal{A}$ else return "wrong guess."
- Else record $\langle \mathsf{offline}, \mathsf{S}, uid, pw^* \rangle$

Password Authentication

– On $(\mathsf{CltSession}, \mathsf{sid}, \mathsf{S}, \boxed{\mathsf{uid}}, pw)$ from C, if there is no record $\langle \mathsf{sid}, \mathsf{C}, ... \rangle$ then save $\langle \mathsf{sid}, \mathsf{C}, \mathsf{S}, \boxed{\mathsf{uid}}, pw, \mathtt{cl} \rangle$ marked fresh, send $(\mathsf{CltSession}, \mathsf{sid}, \mathsf{C}, \mathsf{S}, \boxed{\mathsf{uid}})$ to $\mathcal{A}$.

– On $(\mathsf{SvrSession}, \mathsf{sid}, \mathsf{C}, \mathsf{uid})$ from S, if there is no record $\langle \mathsf{sid}, \mathsf{S}, ... \rangle$ then retrieve record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$, and if it exists then save $\langle \mathsf{sid}, \mathsf{S}, \mathsf{C}, \boxed{\mathsf{uid}}, pw, \mathtt{sr} \rangle$ marked fresh and send $(\mathsf{SvrSession}, \mathsf{sid}, \mathsf{S}, \mathsf{C}, \mathsf{uid})$ to $\mathcal{A}$.

Active Session Attacks

– On $(\mathsf{TestPwd}, \mathsf{sid}, \mathsf{P}, \boxed{\mathsf{uid}}, pw^*)$ from $\mathcal{A}$, if $\exists$ record $\langle \mathsf{sid}, \mathsf{P}, \mathsf{P}', \boxed{\mathsf{uid}}, pw, \mathsf{role} \rangle$ marked fresh, then do: If $pw^* = pw$ then mark it $\mathsf{compromised}$ and return "correct guess" to $\mathcal{A}$; else mark it $\mathsf{interrupted}$ and return "wrong guess."

– On $(\mathsf{Impersonate}, \mathsf{sid}, \mathsf{C}, \mathsf{S}, \mathsf{uid})$ from $\mathcal{A}$, if $\exists$ record $\mathsf{rec} = \langle \mathsf{sid}, \mathsf{C}, \mathsf{S}, \boxed{\mathsf{uid}}, pw, \mathtt{cl} \rangle$ marked fresh, then do: If $\exists$ record $\langle \mathsf{file}, \mathsf{S}, \mathsf{uid}, pw \rangle$ marked $\mathsf{compromised}$ then mark rec $\mathsf{compromised}$ and return "correct guess" to $\mathcal{A}$; else mark it $\mathsf{interrupted}$ and return "wrong guess."

Key Generation and Authentication

– On $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{P}, K^*)$ from $\mathcal{A}$, if $\exists$ record $\mathsf{rec} = \langle \mathsf{sid}, \mathsf{P}, \mathsf{P}', \boxed{\mathsf{uid}}, pw, \mathsf{role} \rangle$ not marked $\mathsf{completed}$, then do:

1. If rec is marked $\mathsf{compromised}$ set $K \leftarrow K^*$;
2. Else if rec is fresh and there is record $\langle \mathsf{sid}, \mathsf{P}', \mathsf{P}, \boxed{\mathsf{uid}}, pw, \mathsf{role}' \rangle$ for $\mathsf{role}' \neq \mathsf{role}$ and $\mathcal{F}_{\mathsf{aPAKE}}$ sent (sid, K') to P' when this record was fresh, set $K \leftarrow K'$;
3. Else set $K \leftarrow_{\mathrm{R}} \{0,1\}^{\ell}$.

Finally, mark rec as $\mathsf{completed}$ and send output (sid, K) to P.

Note: Modifications from $\mathcal{F}_{\mathsf{aPAKE}}$ defined in [25] are marked $\boxed{\textit{like this}}$*. They consist of assumping input* uid *in* $\mathsf{CltSession}$ *and* $\mathsf{TestPwd}$ *and enforcing* uid*-equality between client and server sessions in* NewKey *processing.*

Fig. 9. $\mathcal{F}_{\mathsf{aPAKE}}$: asymmetric PAKE functionality adapted from [25]

Queries StorePwdFile from S, StealPwdFile or OfflineTestPwd from $\mathcal{A}$, CltSession from C, SvrSession from S, and TestPwd or Impersonate from $\mathcal{A}$, functionality $\mathcal{F}_{\text{aPAKE-cEA}}$ acts as $\mathcal{F}_{\text{aPAKE}}$ of Figure 9, *except* it omits all parts marked uid (i.e. it does not require uid input for C and does not enforce uid-equality for C and S).

Below we mark like this parts of NewKey processing which differ from $\mathcal{F}_{\text{aPAKE}}$.

Key Generation and Authentication

- On $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{P}, K^*)$ from $\mathcal{A}$, if there is a record $\mathsf{rec} = \langle \mathsf{sid}, \mathsf{P}, \mathsf{P}', pw, \mathsf{role} \rangle$ not marked completed, then do:
  1. If rec is marked compromised set $K \leftarrow K^*$;
  2. Else if rec is fresh , role = sr, and there is record $\langle \mathsf{sid}, \mathsf{P}', \mathsf{P}, pw, \texttt{cl} \rangle$ s.t. $\mathcal{F}_{\text{aPAKE-cEA}}$ sent (sid, K') to P' when this record was fresh, set $K \leftarrow K'$;
  3. Else if role = cl set $K \leftarrow_{\text{R}} \{0,1\}^{\ell}$, and if role = sr set $K \leftarrow \perp$.

  Finally, mark rec as completed and send output (sid, K) to P.

Fig. 10. $\mathcal{F}_{\text{aPAKE-cEA}}$: asymmetric PAKE with explicit C-to-S authentication

defined by Gentry, Mackenzie, and Ramzan [24], but it adopts few notational modifications introduced by Gu et al. [25]. These include naming what amounts to user accounts explicitly as uid instead of generic-sounding sid, using sid instead of ssid as a session-identifier for on-line authentication attempts, and using only pairs (S, uid) to identify server password files and not (S, U, uid) tuples as in [24].

Because in this paper we differentiate between unsalted and (publicly) salted aPAKE's, an explicit support for unsalted aPAKE's is reflected in aPAKE functionality $\mathcal{F}_{\text{aPAKE}}$ by introducing a slight modification in the functionality of [25]. These modifications are highlighted in Figure 9, and they all concern a client-side usage of the user account field uid. As we mention in the introduction, the round-minimal protocol aEKE is *unsalted*, and to enforce the aPAKE contract defined by [24], which is that a single real-world offline dictionary attack operation must correspond not only to a single password guess but also to a unique user password file, identified by a unique pair (S, uid), the client must get as environment's inputs both the server identifier S *and* the user account identifier uid. This is reflected in including uid in the inputs to CltSession command in Figure 9. However, since the client now performs computation on a fixed uid, honest client and server sessions will not agree on the same output key unless they run not only on the same password pw but also on the same uid. Hence the NewKey processing now includes uid-equality enforcement. Finally, for the same reason, an online password test TestPwd must specify the uid field in addition to password guess pw^*.

Functionality $\mathcal{F}_{\text{aPAKE}}$ currently allows both the server and the client sessions to leak the account identifier uid input to the adversary. The server-side leakage of this information was inherent (although not immediate to observe) in the original aPAKE functionality of [24], and it was adopted by subsequent works, including e.g. [30, 25]. Now, however, we also introduce client-side leakage of the

same information. The uid has to be transmitted from the client to the server before the protocol starts, but it is not clear that the cryptographic protocol should leak it. We leave plugging this leakage and/or verifying whether it is necessary in known aPAKEs, including ours, to future work.

Client-to-server entity authentication. Since our protocol OKAPE shown includes client-to-server authentication (it is *not* optional, and the protocol is insecure without it), it realizes an aPAKE functionality amended by client-to-server entity authentication. We use $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ to denote the variant of aPAKE functionality with uni-directional client-to-server entity authentication, and we include it in Figure 10. Since protocol OKAPE is a *salted* aPAKE, it does not need the uid input on the client side, so the $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ functionality in Figure 10 incorporates all the code of functionality $\mathcal{F}_{\mathsf{aPAKE}}$ but without the uid-related modifications. To simplify NewKey processing functionality $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ in Figure 10 assumes that the client party terminates first, so if two honest parties are connected then the client party computes its session key output first, and it is always the server party which can potentially get the same key copied by the functionality. One could define it more generally but we expect that in most aPAKE protocols with unilateral client-to-server explicit authentication the server will indeed be the last party to terminate.

B Simulator for proof of Theorem 3

Because of space constraints, we refer the reader to [23] for a complete proof of Theorem 3, and provide here an abridged version containing only the overall proof strategy and the description of the simulator.

To prove the theorem we need to construct a simulator, denoted SIM, such that the environment's view of the real-world security game, i.e. an interaction between the adversary $\mathcal{A}$ (whom we consider as a subprocedure of the environment $\mathcal{Z}$) and honest parties following protocol OKAPE, is indistinguishable from the environment's view in the ideal-world interaction between $\mathcal{A}$, SIM, and the functionality $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$.

Simulator construction. We show an overview of our simulation strategy in Fig 11, which gives the top-level view of the real world execution compared to the ideal world execution which involves the simulator SIM shown in Figures 12-13 as well as the simulator $\mathsf{SIM}_{\mathsf{AKE}}$ for the otkAKE subprotocol. The description of simulator SIM is split into two parts as follows: Figure 12 contains the SIM pt.1 part of the diagram in Fig 11, i.e. it deals with adversary's ideal cipher and hash queries, and in addition with the compromise of password files. Figure 13 contains the SIM pt.2 part of the diagram in Fig 11 dealing with on-line aPAKE sessions. We rely on the fact that protocol otkAKE realizes functionality $\mathcal{F}_{\mathsf{otkAKE}}$, so we can assume that there exists a simulator $\mathsf{SIM}_{\mathsf{AKE}}$ which exhibits this UC-security of otkAKE. Our simulator SIM uses simulator $\mathsf{SIM}_{\mathsf{AKE}}$ as a sub-procedure. Namely, SIM hands over to $\mathsf{SIM}_{\mathsf{AKE}}$ the simulation of all C-side and S-side AKE instances

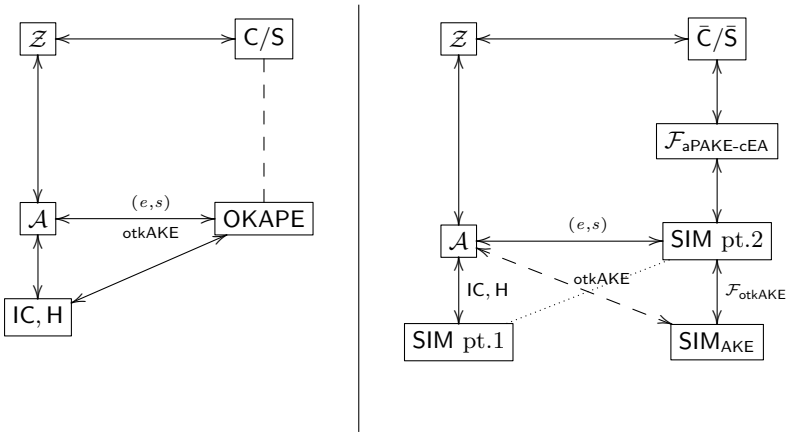


Fig. 11. real-world (left) vs. simulation (right) for protocol OKAPE

where parties run on either honestly generated or adversarial AKE keys. SIM employs SIM_{AKE} to generate such keys - in H queries, password file compromise and in IC decryption queries - see Figure 12, and then it hands off to SIM_{AKE} the handling of all AKE instances that run on such keys, see Figure 13.

Initialization

Initialize simulator $\mathsf{SIM}_{\mathsf{AKE}}$, empty tables T_{IC} and T_{H}, empty lists PK, CPK
Notation: $\mathsf{T}^h_{\mathsf{IC}}.X' = \{x' \mid \exists y\ (h, x', y) \in \mathsf{T}_{\mathsf{IC}}\}$, $\mathsf{T}^h_{\mathsf{IC}}.Y = \{y \mid \exists x'\ (h, x', y) \in \mathsf{T}_{\mathsf{IC}}\}$.
Convention: First call to $\mathsf{SvrSession}$ or $\mathsf{StealPwdFile}$ for $(\mathsf{S}, \mathsf{uid})$ sets $s^{\mathsf{uid}}_{\mathsf{S}} \leftarrow_{\mathrm{R}} \{0,1\}^{\kappa}$.

On query (pw, s) to random oracle H

send back (h, a) if $\exists\ \langle (pw, s), (h, a) \rangle \in \mathsf{T}_{\mathsf{H}}$, otherwise do:

1. If $s \neq s^{\mathsf{uid}}_{\mathsf{S}}$ for all $(\mathsf{S}, \mathsf{uid})$ then $h \leftarrow_{\mathrm{R}} \{0,1\}^{\kappa}$, init. key A via $(\mathsf{Init}, \mathsf{clts}, \mathtt{cl})$ call to $\mathsf{SIM}_{\mathsf{AKE}}$, send $(\mathsf{Compromise}, A)$ to $\mathsf{SIM}_{\mathsf{AKE}}$, define a as $\mathsf{SIM}_{\mathsf{AKE}}$'s response, add A to CPK
2. If $s = s^{\mathsf{uid}}_{\mathsf{S}}$ for some $(\mathsf{S}, \mathsf{uid})$ send $(\mathsf{OfflineTestPwd}, \mathsf{S}, \mathsf{uid}, pw)$ to $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ and:
   (a) if $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ sends "correct guess" then set $A \leftarrow A^{\mathsf{uid}}_{\mathsf{S}}$ and $h \leftarrow h^{\mathsf{uid}}_{\mathsf{S}}$
   (b) else inititalize key A via call $(\mathsf{Init}, \mathsf{clts}, \mathtt{cl})$ to $\mathsf{SIM}_{\mathsf{AKE}}$, add A to PK, pick $h \leftarrow \{0,1\}^{\kappa}$

   In either case send $(\mathsf{Compromise}, A)$ to $\mathsf{SIM}_{\mathsf{AKE}}$, define a as $\mathsf{SIM}_{\mathsf{AKE}}$'s response, add A to CPK, set $\mathsf{info}^{\mathsf{uid}}_{\mathsf{S}}(pw) \leftarrow (A, h)$

In all cases add $\langle (pw, s), (h, a) \rangle$ to T_{H} and send back (h, a)

Ideal Cipher IC queries

- On query (h, x') to $\mathsf{IC.E}$, send back y if $(h, x', y) \in \mathsf{T}_{\mathsf{IC}}$, otherwise pick $y \leftarrow_{\mathrm{R}} Y \setminus \mathsf{T}^h_{\mathsf{IC}}.Y$, add (h, x', y) to T_{IC}, and send back y
- On query (h, y) to $\mathsf{IC.D}$, send back x' if $(h, x', y) \in \mathsf{T}_{\mathsf{IC}}$. Otherwise if there exists $(\mathsf{S}, \mathsf{uid})$ and (A, pw) such that $y = e^{\mathsf{sid}}_{\mathsf{S},\mathsf{uid}}$ and $\mathsf{info}^{\mathsf{uid}}_{\mathsf{S}}(pw) = (A, h)$ then set $\mathsf{id} = \mathsf{S}$, else set $\mathsf{id} = \mathsf{null}$. Initialize key B via call $(\mathsf{Init}, \mathsf{id}, \mathtt{sr})$ to $\mathsf{SIM}_{\mathsf{AKE}}$ and add B to PK. Set $x' \leftarrow_{\mathrm{R}} \mathsf{map}(B)$, add (h, x', y) to T_{IC} and send back x'

Stealing Password Data

On $\mathcal{Z}$'s permission to do so send $(\mathsf{StealPwdFile}, \mathsf{S}, \mathsf{uid})$ to $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$. If $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ sends "no password file," pass it to $\mathcal{A}$, otherwise do the following:

1. if $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ returns pw, set $(A, h) \leftarrow \mathsf{info}^{\mathsf{uid}}_{\mathsf{S}}(pw)$
2. else init. A via call $(\mathsf{Init}, \mathsf{clts}, \mathtt{cl})$ to $\mathsf{SIM}_{\mathsf{AKE}}$, add A to PK, pick $h \leftarrow \{0,1\}^{\kappa}$

Set $(A^{\mathsf{uid}}_{\mathsf{S}}, h^{\mathsf{uid}}_{\mathsf{S}}) \leftarrow (A, h)$, return $\mathsf{file}[\mathsf{uid}, \mathsf{S}] \leftarrow (A^{\mathsf{uid}}_{\mathsf{S}}, h^{\mathsf{uid}}_{\mathsf{S}}, s^{\mathsf{uid}}_{\mathsf{S}})$ to $\mathcal{A}$.

Fig. 12. Simulator SIM showing that protocol OKAPE realizes $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$: Part 1

Starting AKE sessions

On $(\mathsf{SvrSession}, \mathsf{sid}, \mathsf{S}, \mathsf{C}, \mathsf{uid})$ from $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$, initialize random function $R^{\mathsf{sid}}_{\mathsf{S}} : (\{0,1\}^*)^3 \rightarrow \{0,1\}^\kappa$, pick $e^{\mathsf{sid}}_{\mathsf{S},\mathsf{uid}} \leftarrow_{\mathrm{R}} Y$, set $\mathsf{flag}(\mathsf{S}^{\mathsf{sid}}) \leftarrow \mathsf{hbc}$, send $(e^{\mathsf{sid}}_{\mathsf{S},\mathsf{uid}}, s^{\mathsf{uid}}_{\mathsf{S}})$ to $\mathcal{A}$ as a message from $\mathsf{S}^{\mathsf{sid}}$, and send $(\mathsf{NewSession}, \mathsf{sid}, \mathsf{S}, \mathsf{C}, \mathtt{sr})$ to $\mathsf{SIM}_{\mathsf{AKE}}$

On $(\mathsf{CltSession}, \mathsf{sid}, \mathsf{C}, \mathsf{S})$ from $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ and message (e', s') sent by $\mathcal{A}$ to $\mathsf{C}^{\mathsf{sid}}$, initialize random function $R^{\mathsf{sid}}_{\mathsf{C}} : (\{0,1\}^*)^3 \rightarrow \{0,1\}^\kappa$, and:

1. If $\exists$ uid s.t. $(e', s') = (e^{\mathsf{sid}}_{\mathsf{S},\mathsf{uid}}, s^{\mathsf{uid}}_{\mathsf{S}})$, set $\mathsf{flag}(\mathsf{C}^{\mathsf{sid}}) \leftarrow \mathsf{hbc}^{\mathsf{uid}}_{\mathsf{S}}$, go to 5.
2. If $\exists\ x', \mathsf{uid}$ s.t. $s' = s^{\mathsf{uid}}_{\mathsf{S}}$ and e' was output by $\mathsf{IC.E}$ on $(h^{\mathsf{uid}}_{\mathsf{S}}, x')$, send $(\mathsf{Impersonate}, \mathsf{sid}, \mathsf{C}, \mathsf{S}, \mathsf{uid})$ to $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ and:
   (a) If $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ returns "correct guess", $\mathsf{flag}(\mathsf{C}^{\mathsf{sid}}) \leftarrow (\mathsf{act}^{\mathsf{uid}}_{\mathsf{S}}, A^{\mathsf{uid}}_{\mathsf{S}}, \mathsf{map}^{-1}(x'))$
   (b) If it returns "wrong guess", set $\mathsf{flag}(\mathsf{C}^{\mathsf{sid}}) \leftarrow \mathsf{rnd}$.
   Either case, go to 5.
3. If $\exists\ (x', h, a, pw)$ s.t. e' was output by $\mathsf{IC.E}$ on (h, x') and $\langle (pw, s'), (h, a) \rangle \in \mathsf{T_H}$ (SIM aborts if tuple not unique), send $(\mathsf{TestPwd}, \mathsf{sid}, \mathsf{C}, pw)$ to $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ and:
   (a) If $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$ returns "correct guess", $\mathsf{flag}(\mathsf{C}^{\mathsf{sid}}) \leftarrow (\mathsf{act}^{\mathsf{uid}}_{\mathsf{S}}, A, \mathsf{map}^{-1}(x'))$ where A is the public key generated from a.
   (b) If it returns "wrong guess", set $\mathsf{flag}(\mathsf{C}^{\mathsf{sid}}) \leftarrow \mathsf{rnd}$.
   Either case, go to 5.
4. In all other cases set $\mathsf{flag}(\mathsf{C}^{\mathsf{sid}}) \leftarrow \mathsf{rnd}$, go to 5.
5. Send $(\mathsf{NewSession}, \mathsf{sid}, \mathsf{C}, \mathsf{S}, \mathtt{cl})$ to $\mathsf{SIM}_{\mathsf{AKE}}$

Responding to $\mathsf{SIM}_{\mathsf{AKE}}$ messages to $\mathcal{F}_{\mathsf{otkAKE}}$ emulated by SIM

SIM passes otkAKE protocol messages between $\mathsf{SIM}_{\mathsf{AKE}}$ and $\mathcal{A}$, but when $\mathsf{SIM}_{\mathsf{AKE}}$ outputs queries to (what $\mathsf{SIM}_{\mathsf{AKE}}$ thinks is) $\mathcal{F}_{\mathsf{otkAKE}}$, SIM reacts as follows:

If $\mathsf{SIM}_{\mathsf{AKE}}$ outputs $(\mathsf{Interfere}, \mathsf{sid}, \mathsf{S})$ set $\mathsf{flag}(\mathsf{S}^{\mathsf{sid}}) \leftarrow \mathsf{act}$

If $\mathsf{SIM}_{\mathsf{AKE}}$ outputs $(\mathsf{Interfere}, \mathsf{sid}, \mathsf{C})$ and $\mathsf{flag}(\mathsf{C}^{\mathsf{sid}}) = \mathsf{hbc}^{\mathsf{uid}}_{\mathsf{S}}$ then set $\mathsf{flag}(\mathsf{C}^{\mathsf{sid}}) \leftarrow \mathsf{rnd}$

If $\mathsf{SIM}_{\mathsf{AKE}}$ outputs $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{C}, \alpha)$:

1. If $\mathsf{flag}(\mathsf{C}^{\mathsf{sid}}) = (\mathsf{act}^{\mathsf{uid}}_{\mathsf{S}}, A, B)$ then $k \leftarrow R^{\mathsf{sid}}_{\mathsf{C}}(A, B, \alpha)$, output $\tau \leftarrow \mathsf{prf}(k, 1)$ and send $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{C}, \mathsf{prf}(k, 0))$ to $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$
2. Else output $\tau \leftarrow_{\mathrm{R}} \{0,1\}^\kappa$ and send $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{C}, \perp)$ to $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$

If $\mathsf{SIM}_{\mathsf{AKE}}$ outputs $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{S}, \alpha)$ and $\mathcal{A}$ sends τ' to $\mathsf{S}^{\mathsf{sid}}$:

1. If $\mathsf{flag}(\mathsf{S}^{\mathsf{sid}}) = \mathsf{hbc}$ and τ' was generated by SIM for $\mathsf{C}^{\mathsf{sid}}$ s.t. $\mathsf{flag}(\mathsf{C}^{\mathsf{sid}}) = \mathsf{hbc}^{\mathsf{uid}}_{\mathsf{S}}$, then send $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{S}, \perp)$ to $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$
2. If $\mathsf{flag}(\mathsf{S}^{\mathsf{sid}}) = \mathsf{act}$ and $\exists\ (pw, B)$ s.t. $\tau' = \mathsf{prf}(k,1)$ for $k = R^{\mathsf{sid}}_{\mathsf{S}}(B, A, \alpha)$ where $(A, h) = \mathsf{info}^{\mathsf{uid}}_{\mathsf{S}}(pw)$ and $(h, \mathsf{map}(B), e^{\mathsf{sid}}_{\mathsf{S},\mathsf{uid}}) \in \mathsf{T_{IC}}$ (SIM aborts if tuple not unique), send $(\mathsf{TestPwd}, \mathsf{sid}, \mathsf{S}, pw)$ and $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{S}, \mathsf{prf}(k, 0))$ to $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$
3. In any other case send $(\mathsf{TestPwd}, \mathsf{sid}, \mathsf{S}, \perp)$ and $(\mathsf{NewKey}, \mathsf{sid}, \mathsf{S}, \perp)$ to $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$

If $\mathsf{SIM}_{\mathsf{AKE}}$ outputs $(\mathsf{SessionKey}, \mathsf{sid}, \mathsf{P}, pk, pk', \alpha)$:
If $pk' \notin (PK \setminus CPK)$ send $R^{\mathsf{sid}}_{\mathsf{P}}(pk, pk', \alpha)$ to $\mathcal{A}$

Fig. 13. Simulator SIM showing that protocol OKAPE realizes $\mathcal{F}_{\mathsf{aPAKE\text{-}cEA}}$: Part 2