Round-Reduced Modular Construction of Asymmetric Password-Authenticated Key Exchange

Jung Yeon Hwang[1], Stanislaw Jarecki[2], Taekyoung Kwon[3], Joohee Lee[4(✉)], Ji Sun Shin[5], and Jiayu Xu[2]

[1] Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea
videmot@etri.re.kr
[2] University of California, Irvine, USA
{sjarecki,jiayux}@uci.edu
[3] Yonsei University, Seoul, Republic of Korea
taekyoung@yonsei.ac.kr
[4] Seoul National University, Seoul, Republic of Korea
skfro6360@snu.ac.kr
[5] Sejong University, Seoul, Republic of Korea
jsshin@sejong.ac.kr

Abstract. Password-Authenticated Key Exchange (PAKE) establishes a shared key between two parties who hold the same password, assuring security against offline password-guessing attacks. The *asymmetric* PAKE (a.k.a. *augmented* or *verifier-based* PAKE) strengthens this notion by allowing one party, typically a server, to hold a one-way hash of the password, with the property that a compromise of the server allows the adversary to recover the password only via the *offline dictionary attack* against this hashed password. Today's client-to-server Internet authentication is asymmetric, with the server holding only a (salted) password hash, but it relies on client's trust in the server's public key certificate. By contrast, cryptographic PAKE literature addresses the password-only setting, without assuming certified public keys, but it commonly does not address the asymmetric PAKE setting which is required for client-to-server authentication.

The asymmetric PAKE (aPAKE) was defined in the Universally Composable (UC) framework by the work of Gentry et al. [15], who also provided a generic method of converting a UC PAKE to UC aPAKE, at the cost of two additional communication rounds. Motivated by practical applications of aPAKEs, in this paper we propose alternative methods for converting a UC PAKE to UC aPAKE, which use only one additional round. Moreover, since this extra message is sent from client to server, it does not add any round overhead in applications which require explicit client-to-server authentication. Importantly, this round-complexity reduction in the compiler comes at virtually no cost, since with respect to local computation and security assumptions our constructions are comparable to that of Gentry et al. [15].

© Springer Nature Switzerland AG 2018
D. Catalano and R. De Prisco (Eds.): SCN 2018, LNCS 11035, pp. 485–504, 2018.
https://doi.org/10.1007/978-3-319-98113-0_26

Keywords: Communication · Password · Authentication
Key exchange

1 Introduction

Symmetric PAKE and Its Limitations. In the cryptographic literature password authentication is modeled as a Password-Authenticated Key Exchange (PAKE) [4,5,9], a protocol which allows two parties who share only a password to establish a shared cryptographic session key. The main challenge in designing a secure PAKE is the fact that passwords have low entropy and are therefore subject to so-called *dictionary attacks*, a.k.a. *password guessing attacks*, where the adversary searches a moderate-sized dictionary from which the user's password is typically chosen. Every password-authentication protocol is subject to *on-line guessing attacks*, where the adversary runs the prescribed PAKE protocol on a password guess with either the client or the server, and succeeds if its guess was correct. While such attack is unavoidable, its effect can be reduced by limiting the number of unsuccessful authentication session each party is willing to run. However, a PAKE protocol must be secure against an *off-line dictionary attack*, i.e. no efficient adversary can verify any password guess without the on-line interaction described above. Informally, a PAKE protocol is secure if a successful on-line guessing attack is the only way to learn information about the established session keys.

The PAKE security model was introduced by Bellovin and Merritt [5] and was formalized by Bellare *et al.* [4] and Boyko *et al.* [9] via a game-based definition, and then by Canetti *et al.* [12], who formalized PAKE in the Universally Composable (UC) framework [11]. The UC definition of PAKE has become a de facto standard in the cryptographic literature on PAKEs because it is widely recognized as capturing several security issues pertinent to PAKEs which the game-based PAKE notions of [4,9] do not cover. Specifically, apart of standard UC guarantee of security under arbitrary protocol composition, UC PAKE implies *forward-security*, i.e. security of past protocol sessions in case of password compromise, and security for *arbitrary password distribution*, which implies security for *password mistyping* and for *related passwords*.

Most of cryptographic PAKE literature focuses on the *symmetric* PAKE setting, where both parties hold the password. However, if the client-to-server password authentication was implemented with a symmetric PAKE, a compromise of the server would leak the passwords of all the users who authenticate to that server. By contrast, the standard Internet password authentication, password-over-TLS, works in an *asymmetric* setting, where the server holds only a (randomized) one-way hash of the password, and if an adversary compromises the server, the only way it can recover any user's password is by mounting an exhaustive *off-line dictionary attack* using an exhaustive search over some implicit password dictionary, and the attack succeeds only on the passwords which this dictionary included. While this level of protection is far from perfect, as many users choose passwords with too low entropy, it still raises the bar

for the attacker, and protects at least those users whose passwords are hard to guess. This security advantage of an asymmetric password authentication essentially makes symmetric PAKEs not applicable to the client-server setting. On the other hand, the password-over-TLS authentication has weaknesses as well. First, TLS relies on integrity of PKI, and breaks down under various PKI attacks, e.g. human-engineering *phishing* attacks where the user is tricked to authenticate to a malicious site. Secondly, while the server does not permanently store the user's password in the clear, it does hold it in the clear during an authentication session, which makes the password vulnerable to server-side insider attacks, virus attacks, and insecure memory and storage management.

State of Knowledge on Asymmetric PAKE. Cryptographic PAKE literature recognized the need to bridge between the password-authentication theory, i.e. the symmetric but PKI-independent PAKE model, and the password-authentication practice, i.e., the security requirements of client-to-server authentication. The first formalization of *asymmetric PAKE* (*aPAKE*), a.k.a. *augmented* or *verifier-based* PAKE, was introduced by Bellovin and Merritt [6] and formalized in the game-based approach by Boyko *et al.* [9]. Subsequently, Gentry *et al.* [15] extended the UC PAKE model of [12] to the case of an adaptive server compromise, and forcing the adversary to stage an off-line dictionary attack to recover the password after such compromise. While several aPAKE protocols were proven in game-based models, some argued only informally, e.g. [2,7,9,10,22–24], the UC aPAKE notion is stronger than game-based aPAKE for the same reasons that UC PAKE notion is stronger than game-based PAKE, thus ideally we would like to know protocols which realize the UC asymmetric PAKE notion of [9] and are comparable in efficiency and cryptographic assumptions to standard authenticated key agreement protocols used in TLS.

However, there is not much known about provably secure UC aPAKEs. One construction is the Ω-method due to Gentry *et al.* [15], shown in Fig. 1, which transforms any UC PAKE protocol into a UC aPAKE secure in the Random Oracle Model (ROM). The Ω-method compiler adds (up to) two communication rounds to the underlying PAKE, and its computation overhead is dominated by a signature generation for the client and signature verification for the server. Instantiated with ECDSA signatures, both these costs are only 1 (multi-)exponentiation per party.

However, since the Ω-method is a compiler, the exact costs of UC aPAKE it produces depend on the costs of the UC PAKE with which it is instantiated. While there is very active research on standard-model UC PAKEs, including round-minimal PAKEs [14,17,20,21], these constructions are typically more expensive and require stronger assumptions than protocols satisfying game-based PAKE notions [4,9] in ROM. Since any UC aPAKE construction requires non-black-box assumptions, it makes sense to instantiate the Ω-method with a UC PAKE secure in ROM. However, while there are many 2-round game-based PAKEs whose cost is close to (intuitively minimal) 2 exponentiations/party of Diffie-Hellman Key Exchange (see e.g. [3] and references therein), we know of

only one UC PAKE with comparable efficiency, by Abdala et al. [1], which relies on the DDH assumption in ROM and Ideal Cipher (IC) models, and uses 3 protocol rounds and 2 exponentiations per party. Combined with the Ω-method of Gentry *et al.* the UC *symmetric* PAKE of [1] implies a UC *asymmetric* PAKE with 5 rounds, 3 exponentiations per party, secure under the DDH assumption in ROM+IC model.

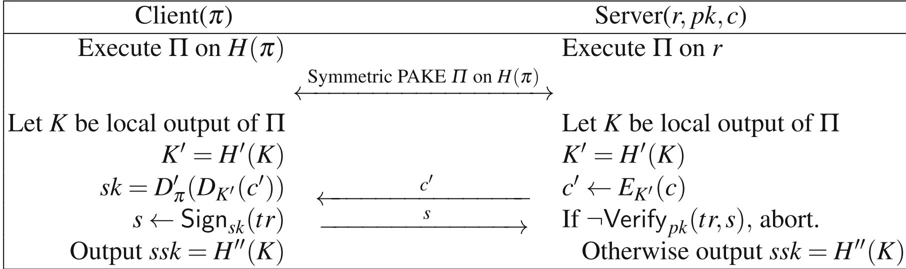

Fig. 1. The Ω-method by Gentry-MacKenzie-Ramzan [15]: H is a hash function, and (E, D) and (E', D') are symmetric encryption schemes (see [15] for their specification). The server-held password file created for password π is (r, pk, c) where $r = H(\pi)$, $c = E'_\pi(sk)$, and (sk, pk) is a private, public key pair in a signature scheme.

We know of only two further UC aPAKE constructions in addition to the Ω-method of [15]. First, Jutla and Roy [19] proposed a round-minimal UC aPAKE in ROM, i.e. client and server send a single message and they can do so simultaneously, but their scheme requires groups with bilinear maps, uses significantly more exponentiations (and bilinear maps) per party. Secondly, Jarecki *et al.* [16] proposed a *strong* UC aPAKE protocol called OPAQUE, where hashed passwords are privately salted (see Sect. 2 for further discussion), which requires only 2 rounds of communication, and only 3 or 4 exponentiations per each party, but it relies on the somewhat non-standard and *interactive* assumption of One-More Diffie-Hellman. This leaves open the possibility of similarly low-round UC aPAKE relying on static assumptions.

Note on Verifier-Based PAKEs. We note that Benhamouda and Pointcheval [7] upgraded the game-based definition of aPAKE, called *verifier-based* PAKE therein, by strengthening the game-based aPAKE model of [9] to arbitrary password distributions and related passwords. One point of strengthening game-based aPAKE notion given that a UC aPAKE notion exists is a potential for better efficiency, but the other is that the UC aPAKE model of Gentry *et al.* [15] seems not to be realizable without some non-black-box assumption on the adversary's local computation, like ROM, IC, or a generic group model. Indeed, the UC aPAKE model [15] requires the simulator to extract off-line password tests from adversary's local computation of the hash function applied to password guesses. However, [7] relies on the *tight one-wayness* requirement on the hash

function applied to passwords when creating the hashed password on the server, namely that given hash of a password chosen with δ min-entropy, the adversary has to compute 2^{δ} hash function instances to find it. Unfortunately, this notion also seems impossible to realize without similar non-black-box assumptions on the adversary, and [7] also rely on ROM to argue that this property is satisfied. Regarding computational costs, by avoiding random oracles on the protocol level (but not on the level of the underlying hash function), the aPAKE's of [9] are significantly more expensive than either the UC aPAKE resulting from [1,15] or the UC aPAKE of [16]. Their 2-round scheme uses significantly more exponentiations per party, and the 1-round scheme requires groups with bilinear maps and has a still higher local computation cost.

Our Results. We show two new compilers which convert any UC-secure *symmetric* PAKE protocol into a UC-secure *asymmetric* PAKE. Our constructions rely on ROM, as do all UC asymmetric PAKE schemes proposed so far [15,16,19], and either the Computational Diffie-Hellman (CDH) or the Discrete Logarithm (DL) assumption. The main point of both compilers is that they add only a *single additional message* to the underlying PAKE, in contrast to the Ω-method of Gentry *et al.* [15] which adds two messages. Moreover, this single extra message is sent from client to server, and therefore in an application where the aPAKE instance, which establishes a secure session key for both parties, is followed by an explicit client-to-server entity authentication, e.g. the client uses the session key output by PAKE to send a MAC on the aPAKE transcript to the server, this additional message can be piggybacked with the client's explicit entity authentication flow. Likewise, if the last message of the symmetric UC PAKE is client-to-server, our compilers also add no additional communication flow to the protocol. By contrast, the Ω-method would add 2 message flows in the latter case.

We note that if the last round in the symmetric UC PAKE was server-to-client, then our compilers would offer no advantage over the Ω-method: Our compilers would add one client-to-server round, and so would the Ω-method because its first message c' (see Fig. 1) would be piggybacked on the last server-to-client flow of the underlying UC PAKE. Moreover, note that the symmetric UC PAKE, by its very nature has no fixed roles, therefore every UC PAKE protocol can be executed so that the last message flow is server-to-client. Indeed, if the underlying n-round UC PAKE is executed in this way then the UC aPAKE resulting from both our compilers and the Ω-method would have $n + 1$ rounds, with the last flow being client-to-server. However, the optimal way to arrange the n-round UC PAKE for the purpose of our compiler is so that its last flow is client-to-server, in which case our compilers output n-round UC aPAKE, while the Ω-method outputs an $(n+2)$-round UC aPAKE. Finally, note that sometimes one will not have the flexibility of arranging the underlying PAKE in a way that optimizes the resulting aPAKE, because sometimes the choice of party who starts the interaction, i.e. whether it is the client or the server, will be fixed by an application.

We show two compilers, one utilizing a parallel round of a Diffie-Hellman Key Exchange, shown in Sect. 3, and one utilizing a NIZK of discrete logarithm knowledge, shown in Sect. 4. We refer to these constructions as respectively CDH-based and DL-based because these are the assumptions they require for security. The computational costs of the first compiler is 1 exponentiation per client and 2 per server, while for the second compiler it is 1 (multi-)exponentiation per both parties, which matches the computational costs of the Ω-method instantiated with ECDSA signature. Looking a little closer, the costs of each option can be affected by the fact that in our DL-based compiler, exactly as in the Ω-method instantiated with ECDSA signature, the client's exponentiation is fixed-base, and therefore can be sped-up by pre-computation, while the server's exponentiation is variable-base, while in the CDH-based compiler the client's exponentiation is variable-base and the two server's exponentiations are fixed-base, with one base fixed globally and the second base fixed per each user account. We summarize this discussion in Table 1 below.

Table 1. Comparison of PAKE-to-aPAKE compiler costs

	Exponentiation cost		Number of added rounds
	Client	Server	
Our CDH-based compiler	1 var. base	2 fixed base	0 or 1
Our DL-based compiler	1 fixed base	1 var. base	0 or 1
Ω method + ECDSA [15]	1 fixed base	1 var. base	1 or 2

Since just like [15] our UC aPAKE constructions are compilers from any UC PAKE, the efficiency and security assumptions of the resulting aPAKE depend also on the underlying UC PAKE. Since our compilers require ROM it makes sense to instantiate them with a low-cost UC PAKE secure in ROM. However, as mentioned above, we know only one UC PAKE constructed along these lines, namely the protocol of Abdalla *et al.* [1]. Because the last message of this PAKE is a client-to-server flow, the UC aPAKE's which result from our compilers applied to UC PAKE of [1] will take 3 rounds and use 3 exponentiations per client and either 3 or 4 exponentiations per server. We include a specification of the UC aPAKE resulting from applying our CDH-based compiler to UC PAKE of [1] in Sect. 5.

We note that theoretically our compiler can also be instantiated with any minimum-round UC PAKE, e.g. [18], but it would result in a 2-round UC aPAKE, and the computation cost of the resulting protocol would be close to (and thus probably not competitive with) the 1-round UC aPAKE of [19].

2 Security Model

Our protocols convert any UC-secure symmetric PAKE into a UC-secure *asymmetric* PAKE, exactly like the protocol of [15], and we assume the same

models of universally composable symmetric PAKE and asymmetric PAKE as in [15], denoted respectively $\mathsf{F}_{\mathsf{rpwKE}}$ and $\mathsf{F}_{\mathsf{apwKE}}$. For completeness we include the full description of both functionalities in Appendix A, in Figs. 5 and 6. Below we sketch the most important points in which these functionalities differ from the standard UC PAKE functionality of [11], and we refer to [15] for their full exposition.

The Revised Symmetric PAKE Functionality [15]**.** The symmetric PAKE functionality $\mathsf{F}_{\mathsf{rpwKE}}$ defined by [15] is a revision of the original PAKE functionality defined by Canetti *et al.* [12]. Namely, it allows the functionality to produce a bitstring representing a *transcript* of the real-world execution of the PAKE protocol. Clearly, every real-world protocol has a transcript, but a typical UC functionality is concerned only with its "functional" input/output behavior and often omits the fact that various "objects" involved in protocol operation, e.g. private keys, public keys, transcripts, have physical encodings as bitstrings. This is unfortunate (and it is often not easy to do) because in protocol composition it can be very useful to process such objects through other cryptographic mechanisms, e.g. to sign them, encrypt them, secret-share them, etc. The idea of the PAKE-to-aPAKE compiler of Gentry *et al.* [15] was for the client to sign the PAKE transcript using a key encrypted by the server using the session key output by the symmetric PAKE. This signature acts in the Gentry *et al.* construction as a proof of possession of the password. However, for this modular construction to work, the UC symmetric PAKE functionality must expose some bitstring as the transcript to the environment. This is the sole point of the revised UC PAKE functionality $\mathsf{F}_{\mathsf{rpwKE}}$ compared to the one defined in [12], and we adopt this revision because our compilers will likewise use the transcript of the symmetric PAKE to bind the proof-of-password-possession to the underlying symmetric PAKE instance, although we will implement this proof-of-password-possession using different cryptographic mechanisms than the encrypted-key/signature-on-transcript protocol of Gentry *et al.*

The Asymmetric PAKE Functionality. The asymmetric functionality $\mathsf{F}_{\mathsf{apwKE}}$ is a more fundamental modification of the symmetric PAKE functionality [12], which models password authentication in the setting where only one party, the client, authenticates using a password, while the other, the server, uses a bitstring called a *password file*, which without loss of generality is an output of some (randomized) one-way function applied to the password during the initialization procedure. For example, in the standard password-over-TLS implementation the password file is a pair consisting of a random nonce known as *salt* and a hash of the password concatenated with this salt value. In the $\mathsf{F}_{\mathsf{apwKE}}$ functionality, Fig. 6, creation of the password file on the server is modeled by command StorePWfile, and note that the server-side invocation of the authentication protocol instance, via command SvrSession, does not take the password as an input because its implicit input is the stored password file corresponding to session ID *sid* of this aPAKE instance. (It is assumed that a unique *sid* would be assigned to each user account held by a given server.)

The other fundamental difference between the asymmetric PAKE functionality $\mathsf{F}_{\mathsf{apwKE}}$ and a symmetric PAKE is that an adversary may adaptively compromise the server and learn the stored password file, which is modeled by query StealPWfile. Such adaptive server compromise allows the adversary to then impersonate the server to the client, modeled via the Impersonate command, because a real-world adversary could use the stolen password file to emulate the server in the authentication protocol. Finally, since the password file is w.l.o.g. an output of some one-way function applied to the password, an adaptive server compromise allows the adversary to stage an off-line dictionary attack: The adversary can compute the same one-way function, a.k.a. *password hash*, on any password guess, which is modeled by the OfflineTestPwd query: If the password file is stolen, this computation allows the adversary to test if its password guess is correct, because then the password hash would match the one in the password file. If the password file is not stolen yet, the adversary can store these pre-computed hashes, which $\mathsf{F}_{\mathsf{apwKE}}$ models by storing the password guesses made by the adversary via the OfflineTestPwd command, and learn if any of these guesses were correct at the moment of server corruption. This is modeled by functionality $\mathsf{F}_{\mathsf{apwKE}}$ checking after the StealPWFile command whether any of the password guesses made via OfflineTestPwd queries is equal to the password used in to create the password file.

Deterministic vs. Salted Hash in Asymmetric PAKE. We note that the above processing of off-line computation of password hashes models asymmetric PAKE protocols where the one-way function used in computation of the password file, a.k.a. *password hash*, is either deterministic or its randomness, a.k.a. *password salt*, is revealed in the protocol. Recently Jarecki *et al.* [16] proposed a strengthening of the UC aPAKE notion of [15] to a *privately salted* UC aPAKE, where the password hash is a randomized function of the password and the randomness stays private until server compromise. This strengthening is modeled by a modified UC aPAKE functionality which allows the adversary to compute relevant password hashes only after server compromise. We note that [16] shows a generic compiler from unsalted or "publicly salted" UC aPAKE, satisfying functionality $\mathsf{F}_{\mathsf{apwKE}}$ which is the target of aPAKE constructions of this paper, to a privately salted UC aPAKE, using an Oblivious Pseudorandom Function (OPRF) scheme. Since the latter can be realized e.g. under the One-More Diffie Hellman assumption using 2 exponentiations for the client and 1 for the server, in ROM, every aPAKE construction satisfying the weaker aPAKE notion of [15], e.g. the aPAKE protocols presented in this paper, implies privately-salted aPAKE satisfying the stronger aPAKE notion of [16], at this modest increase in computational cost.[1]

[1] The compiler of [16] also adds up to 2 extra rounds to the aPAKE protocol, but for example in the case of any of our aPAKE constructions instantiated with the PAKE of Abdalla *et al.* [1] (see Fig. 4), the OPRF instance in the compiler of [16] would be piggybacked with the first two protocol flows, so the resulting *privately salted* UC aPAKE would have the same 3 rounds.

3 Asymmetric PAKE Construction Based on CDH

Our first construction converts a symmetric UC PAKE protocol Π to an asymmetric UC PAKE, just as the compiler of Gentry *et al.* [15], but using a different method.

Our construction, shown in Fig. 2, runs the symmetric PAKE protocol Π on hashed password $r = H_1(\pi)$, but in parallel it also runs a Diffie-Hellman Key Exchange (DH-KE) where the client's contribution is fixed as $V = g^z$ for $z = H_0(\pi)$, i.e. an independent password hash. The server's contribution is $Y = g^y$ for random y is the only message transferred in this DH-KE instance, because the client's contribution $V = g^{H_0(\pi)}$ is part of the password file stored on the server. The key $K_0 = V^y = Y^z = g^{H_0(\pi) \cdot y}$ resulting from this DH-KE could be computed in an off-line dictionary attack given the DH-KE transcript Y, so we hash it together with key K_1 output by symmetric PAKE Π to derive an authenticator $t = H_2(K_0||K_1||[\ldots])$ which is sent from the client to the server before another hash of key K_1 is used as the session key. Note that security of PAKE Π implies that key K_1 is pseudorandom except if the adversary learns $r = H_1(\pi)$ and succeeds in an *on-line* dictionary attack on Π, hence t is safe from *off-line* dictionary attacks.

The role key K_0 plays in the derivation of authenticator t is to force the adversary to perform an off-line attack against password π *after* compromise of the server. Note that protocol Π plays no security role after server compromise because the adversary can then execute the symmetric PAKE Π on the correct input $r = H_1(\pi)$. However, the DH-KE key $K_0 = Y^{H_0(\pi)}$ is pseudorandom unless the adversary queries H_0 on π, an event which the UC simulator (assuming ROM) can catch and identify as an off-line password test. Note that the adversary who learns the server-stored values $r = H_1(\pi)$ and $V = g^{H_0(\pi)}$ can also perform an off-line test by hashing its password guesses via H_0 and H_1, but the point is that our DH-KE instance key K_1 does not offer any *easier* way for the adversary to find a password than an off-line dictionary attack, which is unavoidable in the asymmetric PAKE setting after server compromise.

Detailed Description of CDH-Based aPAKE Construction. The resulting protocol is shown in Fig. 2, and here we go over this construction in more details. Let Π be an arbitrary secure symmetric PAKE protocol, $\mathbb{G}$ be a finite cyclic group of order q, and $g \in \mathbb{G}$ be its generator, and triple $(\mathbb{G}, g, q)$ is a public parameter of the scheme. Let H_0 be a hash function with range $\mathbb{Z}_q$, and let H_1, H_2, H_3 be three independent hash functions with range $\{0,1\}^\ell$ where ℓ is the security parameter.

Password Enrollment. The user's password file stored on the server is a pair (r, V) which is formed given the user's password π as $r = H_1(\pi)$ and $V = g^{H_0(\pi)}$.

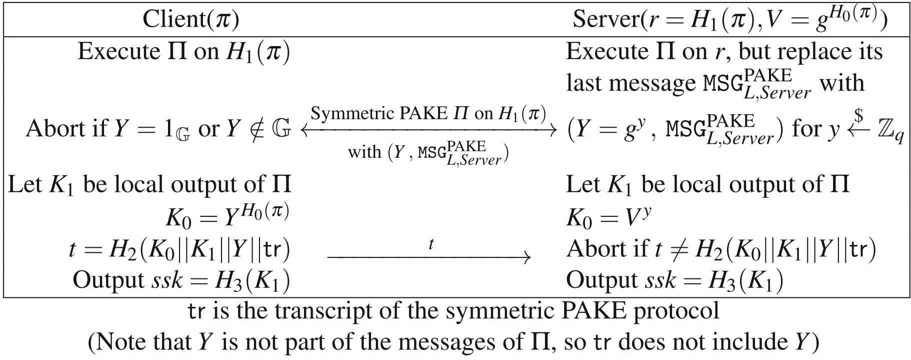

Fig. 2. Construction I: CDH-based compiler from symmetric PAKE to asymmetric PAKE

Protocol Description.

- Client Part 1: The client runs the client-side protocol in the symmetric PAKE Π on input $H_0(\pi)$.
- Server Part 1: The server runs the server-side protocol in the symmetric PAKE Π on input r. In parallel, the server picks a random exponent y in $\mathbb{Z}_q$ and sends $Y = g^y$ to the client along with the last message $\mathtt{MSG}^{\mathrm{PAKE}}_{L,Server}$ of Π. Let K_1 be the server's session key output by protocol Π.
- Client Part 2: Upon receiving message $(Y, \mathtt{MSG}^{\mathrm{PAKE}}_{L,Server})$ from the server, the client aborts if $Y = 1_{\mathbb{G}}$ or $Y \notin \mathbb{G}$. If the check passes, the client completes its Π instance, and let K_1 be the client's session key output by Π. The client computes $K_0 = Y^{H_0(\pi)}$ and $t = H_2(K_1||K_0||Y||\mathsf{tr})$ where tr is the client's transcript of the symmetric PAKE instance Π, sends t to the server, and outputs $ssk = H_3(K_1)$ as its session key.
- Server Part 2: Upon receiving message t from the client, the server computes $K_0 = V^y$ and aborts if $t \neq H_2(K_1||K_0||Y||\mathsf{tr})$ where tr is the server's transcript of the symmetric PAKE instance Π. If the check verifies, the server outputs $ssk = H_3(K_1)$ as its session key.

Cost Discussion. The key import of our compiler construction shown in Fig. 2 is that it adds only one message to the underlying symmetric PAKE Π. Moreover, if the last message of Π is from the client to the server, which happens whenever Π provides explicit entity authentication, then this additional message t can be piggy-backed with the last flow of Π. An example of the latter UC PAKE is the construction of Abdalla *et al.* [1], and as can be seen in Fig. 4 in Sect. 5, our PAKE-to-aPAKE compiler applied to this UC PAKE does not increase its message complexity. By contrast, the same cannot be done in the Ω-method of Gentry *et al.* [15], because it adds two messages, server-to-client and client-to-server to the symmetric PAKE (see Fig. 1), and the server-to-client message can be sent only after the server finalizes the PAKE instance Π because it requires

server's session key output by Π. Therefore, if the final flow of Π is client-to-server the Ω-method would add two messages to the underlying symmetric PAKE protocol Π.

In terms of computational costs, as discussed in the introduction the Ω-method instantiated with ECDSA [25] signatures requires one (multi)exponentiation per party for resp. signature creation and verification, where client's exponentiation is fixed-base and server's (multi)exponentiation is variable-base, while our compiler requires one variable-base exponentiation for the client and two fixed-base exponentiations for the server, for computing $Y = g^y$ and $K_0 = V^y$, although base V is not fixed globally but only per user account.

3.1 Security Argument for Our CDH-Based aPAKE Construction

We state the security of our asymmetric PAKE protocol in Theorem 1 below. In the security argument we model hash functions H_0, H_1, H_2, H_3 used by this protocol as random oracles. For lack of space, we only include an informal sketch of the simulator, and the full formal proof of this theorem is deferred to the full version.

Theorem 1. *If* $(\mathbb{G}, g, q)$ *is a cyclic group in which CDH assumption holds and if protocol* Π *realizes the revised symmetric UC PAKE functionality* $\mathsf{F}_{\mathsf{rpwKE}}$ *then the protocol in Fig. 2 securely realizes the UC aPAKE functionality* $\mathsf{F}_{\mathsf{apwKE}}$ *in the random oracle model for hash functions* H_0, H_1, H_2, H_3.

Simulator Construction. Let A be an adversary that interacts with the parties running the protocol. In the proof we will assume that the execution of the symmetric PAKE protocol Π and calls to hash functions H_0, H_1, H_2, H_3 are replaced by an interaction with, respectively, an ideal functionality $\mathsf{F}_{\mathsf{rpwKE}}$ and ideal functionality F_{RO} modeling (four instances of) a random oracle. We construct a simulator S interacting with the ideal functionality $\mathsf{F}_{\mathsf{apwKE}}$ such that no probabilistic polynomial time environment Z can distinguish an interaction with A in the $(\mathsf{F}_{\mathsf{rpwKE}}, \mathsf{F}_{\mathsf{RO}})$-hybrid real world (henceforth real world) from an interaction with S in the simulated ideal world (henceforth ideal world). Without loss of generality, we assume that A is a "dummy" adversary that merely passes all messages from and to Z, and all computations to Z.

In the following argument, we will use π to denote the original password input to P_j while the password file was stored, i.e. that Z sends $(\mathsf{StorePWfile}, sid, P_i, \pi)$ to P_j, we will use π' to denote the password which P_i uses on its authentication session, i.e. that Z sends $(\mathsf{CltSession}, sid, ssid, P_j, \pi')$ to P_i, and we will use w to denote any other password candidate, e.g. in adversary's off-line password test queries.

S's essential tasks are as follows:

- Dealing with server compromise and offline attacks: When A compromises (sid, P_j), S must come up with r which is supposed to be $H_1(\pi)$, and V which

is supposed to be $g^{H_0(\pi)}$, yet S may not know π. In this case, S chooses r and v (supposed to be $H_0(\pi) = \log_g V$) at random, and whenever S learns π (this occurs when A queries $H_1(\pi)$ or $H_0(\pi)$ and S sends an $\mathsf{OfflineTestPwd}$ message on π to $\mathsf{F}_{\mathsf{apwKE}}$), S "programs" the random oracle results. (If S knows π, it is trivial to compute r and V.)

- P_i's message and output: When P_i's PAKE session is completed and A sends Y', S must come up with t which is supposed to be $H_2(K_1||K_0||Y'||\mathsf{tr})$ and let P_i output its session key which is supposed to be $H_3(K_1)$. t is random to Z unless it knows both K_1 and K_0; P_i's output is random to Z unless it knows K_1. The only way for Z to learn K_1 is via compromising P_i's PAKE session using the correct $r' = H_1(\pi')$, which in turn can be learned via (a) compromising (sid, P_j) to get $r = H_1(\pi)$ and then setting $\pi' = \pi$, or (b) querying $H_1(\pi')$. In case (a), Z also gets $V = g^{H_0(\pi)}$, thus it can compute K_0 as $V^{y'}$ (where $y' = \log Y'$ and can be chosen by Z). In case (b), Z must explicitly query $H_0(\pi')$ in order to learn K_0. In sum, there are four subcases:
  - Case (a) above: S sets t as $H_2(K_1||K_0||Y'||\mathsf{tr})$, and compromises P_i's session in $\mathsf{F}_{\mathsf{apwKE}}$ via an $\mathsf{Impersonate}$ message. Then S is able to set P_i's output, so it sets ssk as $H_3(K_1)$.
  - Case (b) above, and Z queries $H_0(\pi)$: Same as above, except that S compromises P_i's session in $\mathsf{F}_{\mathsf{apwKE}}$ via a $\mathsf{TestPwd}$ message on π', which can be extracted by observing all $H_1(w)$ queries and checking which one was used by A to compromise P_i's PAKE session.
  - Case (b) above, and Z does not query $H_0(\pi)$: Then Z learns K_1, but K_0 is random to it. So S chooses t at random, but still compromises P_i's $\mathsf{F}_{\mathsf{apwKE}}$ session and sets ssk as $H_3(K_1)$.
  - Neither case (a) nor case (b) above holds: Then both K_1 and K_0 are random to Z. So S chooses t at random and does not compromise P_i's session in $\mathsf{F}_{\mathsf{apwKE}}$ (so P_i's output is a random string chosen by $\mathsf{F}_{\mathsf{apwKE}}$).
- P_j's output: When P_j's PAKE session is completed and A sends t', S must let P_j abort, or output its session key which is supposed to be $H_3(K_1)$. As discussed in Sect. 3, P_j aborts unless (a) A merely passes all messages between the client and the server, or (b) A knows both K_1 and K_0, which in turn renders S's ability to extract P_j's password π and use it to compromise P_j's session in $\mathsf{F}_{\mathsf{apwKE}}$. So there are three subcases:
  - Case (a) above: If P_j's PAKE session is compromised, Z knows K_1, so S compromises P_j's session in $\mathsf{F}_{\mathsf{apwKE}}$ and sets ssk as $H_3(K_1)$; otherwise P_j's output is random to Z, so S does not compromise P_j's session in $\mathsf{F}_{\mathsf{apwKE}}$ (so P_j's output is a random string chosen by $\mathsf{F}_{\mathsf{apwKE}}$).
  - Case (b) above: S extracts π as described in Sect. 3 and compromises P_j's session in $\mathsf{F}_{\mathsf{apwKE}}$ via a $\mathsf{TestPwd}$ message on π. Then S sets ssk as $H_3(K_1)$.
  - Neither case (a) nor case (b) above holds: Then S lets P_j abort.

4 Asymmetric PAKE Construction Based on NIZK

Our second aPAKE construction is based on a non-interactive zero-knowledge proof of knowledge (NIZK-PK) of Discrete Logarithm (DL). The construction

is shown in Fig. 3, and it runs an instance of a symmetric PAKE, just like our construction in Sect. 3 and the construction of Gentry *et al.* [15], but here the symmetric PAKE instance is followed by the client sending to the server a NIZK-PK of the password hash $v = H_0(\pi)$, where the verification value $V = g^{H_0(\pi)}$ is held by the server. Transmitting such NIZK-PK in the clear would enable off-line dictionary attacks, so the NIZK-PK should be encrypted under the session key output by the underlying symmetric PAKE. However, a low-cost implementation of this NIZK-PK in ROM via the Fiat-Shamir heuristic [13] can be effectively encrypted if the session key K output by the symmetric PAKE is hashed to derive the verifier's challenge in this NIZK.

Recall that a Fiat-Shamir NIZK-PK of the DL $v = DL(g, V)$ in ROM is implemented by a pair (X, z) s.t. $X = g^x$ for x randomly chosen in $\mathbb{Z}_q$ and $z = x + v \cdot c \bmod q$, where the verifier's challenge c is computed as a hash of X and the DL instance V, i.e. $c = H_2(X||V)$. However, here we modify this challenge-derivation in several ways: First, as mentioned above we include in the hash input key K output by the symmetric PAKE instance Π. This has an effect of encrypting this NIZK proof because if $c = H_2(K||[\ldots])$ then (X, z) is distributed uniformly in $G \times \mathbb{Z}_q$ to an adversary for whom K is pseudorandom. Secondly, we omit the DL challenge V from the hash, to save one exponentiation from the client who would otherwise have to compute it as $V = g^{H_0(\pi)}$. This removal endangers the proof-of-knowledge property of this NIZK, but we replace V with the transcript tr of the symmetric PAKE instance Π. This inclusion has an effect of binding the NIZK to the PAKE instance, including its input $r = H_1(\pi)$, and it suffices to show that the adversary cannot create such NIZK unless it queries H_0 on π or computes the DL $v = DL(g, V)$.

In effect, similarly as in construction of Sect. 3, to generate a valid last message an adversary must either merely pass all messages between the client and the server, or it must know both $v = H_0(\pi)$ and K. To know K, the adversary must learn $H_1(\pi)$ via compromising the server or querying it, and then interfering with the symmetric PAKE using $H_1(\pi)$. If adversary compromises the server and learns $r = H_1(\pi)$, then the NIZKs (X, z) is sees by interacting with the client leak no information because they can be simulated from $V = g^{H_0(\pi)}$. Conversely, if the adversary actively engages the server on any session then it must produce such NIZK-PK itself, and by the similar argument as use in the standard implementation of this proof of knowledge, i.e. when $c = H_2(K||V)$, this is impossible unless adversary either queries H_0 on π or computes the discrete logarithm $v = DL(g, V)$. In each of these cases the simulator can observe what the adversary queries and react respectively.

Detailed Description of NIZK-Based aPAKE Construction. The resulting protocol is shown in Fig. 3, and here we explain it in more details. The setting is exactly the same as in the protocol of Sect. 3 in Fig. 2, except that here H_2 is a hash function onto range $\mathbb{Z}_q$.

Password Enrollment. As in the DH-based construction of Fig. 2, the password file is a pair (r, V) where $r = H_1(\pi)$ and $V = g^{H_0(\pi)}$, where π is the user's password.

Protocol Description.

- Client Part 1: As in the construction of Fig. 2, the client runs the client-side protocol in the symmetric PAKE Π on input $H_0(\pi)$. Let K be the client's session key output by this instance of Π.
- Server Part 1: The server runs the server-side protocol in the symmetric PAKE Π on input r. Let K be the server's session key output by this instance of Π.
- Client Part 2: The client picks random x in $\mathbb{Z}_q$, computes $v = H_0(\pi)$, $X = g^x$, and $z = x + v \cdot c \bmod q$ for $c = H_2(K||X||\mathsf{tr})$ where tr is the client's transcript of Π. The client sends (X, z) to the server, and outputs $ssk = H_3(K)$ as its session key.
- Server Part 2: Upon receiving (X, z) from the client, the server verifies if $X = g^z \cdot V^{-H_2(K||X||\mathsf{tr})}$ where tr is the server's transcript of Π. If the check fails, the server aborts, and otherwise it outputs $ssk = H_3(K)$ as its session key.

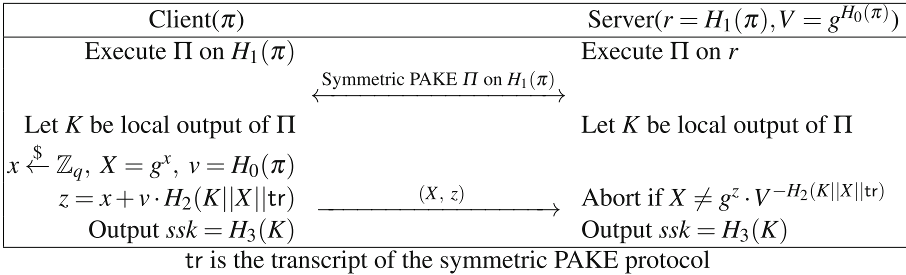

Fig. 3. Construction II: NIZK-based compiler from symmetric PAKE to asymmetric PAKE

Cost Discussion. The added cost of the construction in Fig. 3 is the cost of the NIZK prover and the NIZK verifier. Implemented as above, these require one exponentiation with fixed-base for the client, and one variable-base (multi-)exponentiation for the server. These computational costs are exactly the same as in the Ω-method instantiated with the ECDSA signature, because ECDSA signature is indeed a version of the same NIZK-PK of discrete logarithm as we use here. However, the construction in Fig. 3 has the same communication pattern as the construction in Fig. 2, and hence saves two communication rounds compared to the Ω-method if the last message in the underlying symmetric PAKE goes from the client to the server.

4.1 Security Argument for aPAKE Construction Based on NIZK

As in the first compiler, we state the security of our second compiler as well as a sketch of the simulator below, and defer the full proof to the full version.

Theorem 2. *If* $(\mathbb{G}, g, q)$ *is a cyclic group in which the DL assumption holds and if protocol* Π *realizes the* $\mathsf{F}_{\mathsf{rpwKE}}$ *functionality, then the protocol in Fig. 3 securely realizes the* $\mathsf{F}_{\mathsf{apwKE}}$ *functionality in the random oracle model for hash functions* H_0*,* H_1*,* H_2*,* H_3*.*

Simulator Construction. The simulator S is very similar to that in the proof of Theorem 1. Indeed, since P_j's password file is identical to that in the previous protocol, how S deals with server compromise and offline attacks is exactly the same with the previous simulator.

When P_i's PAKE session is completed, S must come up with X and z, which are supposed to be g^x for x random in $\mathbb{Z}_q$ and $x + H_0(\pi') \cdot H_2(K||X||\mathsf{tr})$, respectively. Value X is a random group element, so S can simply choose it at random; z is random to Z (independent of X) unless Z knows K. As analyzed in Theorem 1, Z gets to know K via either of the following two approaches: (a) compromising (sid, P_j) and then setting $\pi' = \pi$, or (b) querying $H_1(\pi')$. In case (a), S chooses $v = H_0(\pi)$ when A compromises (sid, P_j), so it computes $z = x + v \cdot H_2(K||X||\mathsf{tr})$. In case (b), S computes $z = x + H_0(\pi') \cdot H_2(K||X||\mathsf{tr})$, where $H_0(\pi')$ is chosen at random if undefined yet. Finally, if neither (a) nor (b) holds, K is random to Z, so S chooses z at random.

When P_j's PAKE session is completed and A sends X' and z', S must let P_j abort, or output its session key which is supposed to be $H_3(K)$. P_j aborts unless (a) A merely passes all messages between the client and the server, or

Client $\mathrm{U}(\pi)$		Server $\mathrm{S}(H_1(\pi), V = g^{H_0(\pi)})$
Compute $H_1(\pi)$		$b \xleftarrow{\$} \mathbb{Z}_q^*$, $B = g_1^b$
$a \xleftarrow{\$} \mathbb{Z}_q^*$, $A = g_1^a$	$\xrightarrow{\mathrm{U}, A}$	$y \xleftarrow{\$} \mathbb{Z}_q^*$, $Y = g^y$
Abort if $Y = 1_{\mathbb{G}}$ or $Y \notin \mathbb{G}$	$\xleftarrow{\mathrm{S}, [B^*\|\|Y]}$	$B^* \leftarrow \mathsf{E}_{ssid\|\|H_1(\pi)}(B)$
$B = \mathsf{D}_{ssid\|\|H_1(\pi)}(B^*)$		
$K_U = B^a$		$K_S = A^b$
$Auth = H''(ssid\|\|U\|\|S\|\|A\|\|B\|\|K_U)$		
$K_1 = H'(ssid\|\|U\|\|S\|\|A\|\|B\|\|K_U)$		$K_1 = H'(ssid\|\|U\|\|S\|\|A\|\|B\|\|K_U)$
$K_0 = Y^{H_0(\pi)}$		$K_0 = V^y$
$t = H_2(K_1\|\|K_0\|\|U\|\|A\|\|S\|\|B^*\|\|Y)$	$\xrightarrow{Auth,\ t}$	If $Auth = H''(ssid\|\|U\|\|S\|\|A\|\|B\|\|K_U)$
Output $ssk = H_3(K_1)$		and $t = H_2(K_1\|\|K_0\|\|U\|\|A\|\|S\|\|B^*\|\|Y)$,
		then output $ssk = H_3(K_1)$, else abort.

Fig. 4. A two-round UC asymmetric PAKE using compiler of Sect. 3 instantiated with UC-Secure protocol of [1]. (E, D) is a symmetric encryption modeled as ideal cipher over group G as the message space, g_1 is another generator of the same group G, and H' and H'' are hash functions with range $\{0, 1\}^\ell$.

(b) Z knows K and queries $H_0(\pi)$ (and thus can choose a random x' in $\mathbb{Z}_q$ and set $X' = g^{x'}$ and $z = x' + H_0(\pi) \cdot H_2(K||X'||\mathsf{tr}'))$. Case (a) is similar to the corresponding case in the previous simulator. S can tell case (b) by checking whether $X = g^{z - H_0(\pi) \cdot H_2(K||X'||\mathsf{tr}')}$. If neither (a) nor (b) holds, S forces P_j to abort.

5 An Efficient Instantiation of Our Method

To exemplify the practical implications of our reduced-round PAKE-to-aPAKE compilers, we present a concrete instantiation of our aPAKE construction of Sect. 3 where the PAKE subprotocol is instantiated with the UC PAKE of Abdalla *et al.* [1] (henceforth referred to as the ACCP protocol), which is a variant of the encrypted key exchange (EKE) of Bellovin and Merritt [5]. The ACCP protocol is proven secure in the UC framework under the DDH assumption in the Ideal Cipher (IC) model and ROM, where the Ideal Cipher assumption is posited on a symmetric cipher with cyclic group G as a message space. The ACCP protocol uses three rounds and 2 exponentiations per party. By combining this protocol with our PAKE-to-aPAKE compiler in Fig. 2, we obtain a highly efficient UC *asymmetric* PAKE, depicted in Fig. 4, with the same 3 rounds as the underlying symmetric PAKE protocol, because in the ACCP protocol the last message goes from the client to the server, and therefore our client-to-server message t (see Fig. 2) can be piggybacked onto it. Note that the Ω-method of [15] would instead result in a 5-round protocol, because its two-message interaction, server-to-client and client-to-server, can start only when the server in the underlying symmetric PAKE outputs a session key, which is round 3 in the ACCP protocol.

Cost Discussion. The computational cost of the asymmetric PAKE of Fig. 4 is 3 exponentiations per client (one fixed-base, two variable-base) and 4 per server (three fixed-base, one variable-base), and the cost of the ideal-cipher encryption and decryption. (See e.g. [8,26] on how an ideal cipher can be implemented over an elliptic curve group.) Note that if we instantiate the PAKE-to-aPAKE construction of Fig. 3 with the same ACCP symmetric PAKE protocol, the resulting asymmetric PAKE would have the same communication pattern and 3 exponentiations per client (two fixed-base, one variable-base) and 3 per server (one fixed-base, two variable-base).

Acknowledgements. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government, Ministry of Science and ICT (MSIT) (No. 2016-0-00097, Development of Biometrics-Based Key Infrastructure Technology for Online Identification, and No. 2018-0-01369, Developing blockchain identity management system with implicit augmented authentication and privacy protection for O2O services), and supported by the MSIT, Korea, under the ITRC (Information Technology Research Center) support programs (IITP-2018-0-01423, and IITP-2018-2016-0-00304) supervised by the IITP. This work was also supported by Samsung Research Funding Center of Samsung Electronics under Project (No. SRFC-TB1403-52). We would like to thank anonymous SCN 2018 reviewers for their valuable comments.

A UC Password Authentication Functionalities

For reference we include the specification of functionalities $\mathsf{F}_{\mathsf{rpwKE}}$ and $\mathsf{F}_{\mathsf{apwKE}}$ introduced by [15] for modeling resp. symmetric PAKE and asymmetric PAKE protocols. We refer to Sect. 2 for an overview of these functionalities, and to [15] for their full discussion.

Functionality $\mathsf{F}_{\mathsf{rpwKE}}$

The functionality $\mathsf{F}_{\mathsf{rpwKE}}$ is parameterized by a security parameter ℓ. It interacts with an adversary S and a set of parties via the following queries:

Upon receiving a query $(\mathsf{NewSession}, sid, P_i, P_j, \pi, \mathsf{role})$ **from party** P_i:

Send $(\mathsf{NewSession}, sid, P_i, P_j, \mathsf{role})$ to S. In addition, if this is the first NewSession query, or if this is the second NewSession query and there is a record (P_j, P_i, π'), then record (P_i, P_j, π) and mark it fresh.

Upon receiving a query $(\mathsf{TestPwd}, sid, P_i, \pi')$ **from adversary** S:

If there is a record of the form (P_i, P_j, π) which is fresh, then do: If $\pi = \pi'$, mark the record compromised and reply to S with "correct guess." Otherwise, mark the record interrupted and reply to S with "wrong guess."

Upon receiving a query $(\mathsf{NewKey}, sid, P_i, K)$ **from** S**, where** $|K| = \ell$:

If there is a record of the form (P_i, P_j, π) that is not marked completed, then:

• If this record is compromised, or either P_i or P_j is corrupted, then output (sid, K) to player P_i.

• If this record is fresh, and there is a record (P_j, P_i, π') with $\pi' = \pi$, a key K' was sent to P_j, and (P_j, P_i, π) was fresh at the time, then output (sid, K') to P_i.

• In any other case, pick a random key K'' of length ℓ and sends (sid, K'') to P_i.

Either way, mark the record (P_i, P_j, π) as completed.

Upon receiving a query $(\mathsf{NewTranscript}, sid, P_i, \mathsf{tr})$ **from** S:

If there is a record of the form (P_i, P_j, π) that is marked completed, then:

• If (1) there is a record (P_j, P_i, π') for which tuple $(\mathsf{transcript}, sid, \mathsf{tr}')$ was sent to P_j, (2) either (P_i, P_j, π) or (P_j, P_i, π') was ever compromised or interrupted, and (3) $\mathsf{tr} = \mathsf{tr}'$, ignore the query.

• In any other case, send $(\mathsf{transcript}, sid, \mathsf{tr})$ to P_i.

Fig. 5. The revised symmetric PAKE functionality $\mathsf{F}_{\mathsf{rpwKE}}$ [15]

Functionality $\mathsf{F}_{\mathsf{apwKE}}$

Functionality $\mathsf{F}_{\mathsf{apwKE}}$ is parameterized by a security parameter ℓ. It interacts a set of parties and adversary S via the following queries:

Password storage and authentication sessions

Upon receiving a query $(\mathsf{StorePWfile}, sid, P_i, \pi)$ **from party** P_j:

If this is the first StorePWfile query, store password data record $(\mathsf{file}, P_i, P_j, \pi)$ and mark it uncompromised.

Upon receiving a query $(\mathsf{CltSession}, sid, ssid, P_j, \pi)$ **from party** P_i:

Send $(\mathsf{CltSession}, sid, ssid, P_i, P_j)$ to S, and if this is the first CltSession query for $ssid$, store session record $(ssid, P_i, P_j, \pi)$ and mark it fresh.

Upon receiving a query $(\mathsf{SvrSession}, sid, ssid)$ **from party** P_j:

If there is a password data record $(\mathsf{file}, P_i, P_j, \pi)$, then send $(\mathsf{SvrSession}, sid, ssid, P_i, P_j)$ to S and if this is the first SvrSession query for $ssid$, store session record $(ssid, P_j, P_i, \pi)$, and mark it fresh.

Stealing password data

Upon receiving a query $(\mathsf{StealPWfile}, sid)$ **from adversary** S:

If there is no password data record, reply to S with "no password file." Otherwise, do the following. If the password data record $(\mathsf{file}, P_i, P_j, \pi)$ is marked uncompromised, mark it compromised. If there is a tuple $(\mathsf{offline}, \pi')$ stored with $\pi = \pi'$, send π to S, otherwise reply to S with "password file stolen."

Upon receiving a query $(\mathsf{OfflineTestPwd}, sid, \pi')$ **from adversary** S:

If there is no password data record, or if there is a password data record $(\mathsf{file}, P_i, P_j, \pi)$ that is marked uncompromised, then store $(\mathsf{offline}, \pi')$. Otherwise, do: If $\pi = \pi'$, reply to S with "correct guess." If $\pi \neq \pi'$, reply with "wrong guess."

Active session attacks

Upon receiving a query $(\mathsf{TestPwd}, sid, ssid, P, \pi')$ **from adversary** S:

If there is a session record of the form $(ssid, P, P', \pi)$ which is fresh, then do: If $\pi = \pi'$, mark the record compromised and reply to S with "correct guess." Otherwise, mark the record interrupted and reply to S with "wrong guess."

Upon receiving a query $(\mathsf{Impersonate}, sid, ssid)$ **from adversary** S:

If there is a session record of the form $(ssid, P_i, P_j, \pi)$ which is fresh, then do: If there is a password data record $(\mathsf{file}, P_i, P_j, \pi)$ that is marked compromised, mark the session record compromised and reply to S with "correct guess," else mark the session record interrupted and reply with "wrong guess."

Key generation and authentication

Upon receiving a query $(\mathsf{NewKey}, sid, ssid, P, K)$ **from adversary** S**, where** $|K| = \ell$:

If there is a record of the form $(ssid, P, P', \pi)$ that is not marked completed, then:

- If this record is compromised, or either P or P' is corrupted, then output $(sid, ssid, K)$ to P.
- If this record is fresh, there is a session record $(ssid, P', P, \pi')$, $\pi' = \pi$, a key K' was sent to P', and $(ssid, P', P, \pi)$ was fresh at the time, then let $K'' = K'$, else pick a random key K'' of length ℓ and output $(sid, ssid, K'')$ to P.
- In any other case, pick a random key K'' of length ℓ and output $(sid, ssid, K'')$ to P.

Finally, mark the record $(ssid, P, P', \pi)$ as completed.

Upon receiving a query $(\mathsf{TestAbort}, sid, ssid, P)$ **from adversary** S:

If there is a record of the form $(ssid, P, P', \pi)$ that is not marked completed, then:

- If this record is fresh, there is a record $(ssid, P', P, \pi')$, and $\pi' = \pi$, let $b' = \mathsf{succ}$.
- In any other case, let $b' = \mathsf{fail}$.

Send b' to S. If $b' = \mathsf{fail}$, send $(\mathsf{abort}, sid, ssid)$ to P, and mark record (sid, P, P', π) completed.

Fig. 6. The asymmetric PAKE functionality $\mathsf{F}_{\mathsf{apwKE}}$ [15]

References

1. Abdalla, M., Catalano, D., Chevalier, C., Pointcheval, D.: Efficient two-party password-based key exchange protocols in the UC framework. In: Malkin, T. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 335–351. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79263-5_22
2. Abdalla, M., Chevassut, O., Pointcheval, D.: One-time verifier-based encrypted key exchange. In: Vaudenay, S. (ed.) PKC 2005. LNCS, vol. 3386, pp. 47–64. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30580-4_5
3. Abdalla, M., Pointcheval, D.: Simple password-based encrypted key exchange protocols. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 191–208. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30574-3_14
4. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated key exchange secure against dictionary attacks. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 139–155. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45539-6_11
5. Bellovin, S.M., Merritt, M.: Encrypted key exchange: password-based protocols secure against dictionary attacks. In: IEEE Computer Society Symposium on Research in Security and Privacy - S&P 1992, pp. 72–84. IEEE (1992)
6. Bellovin, S.M., Merritt, M.: Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. In: ACM Conference on Computer and Communications Security - CCS 1993, pp. 244–250. ACM (1993)
7. Benhamouda, F., Pointcheval, D.: Verifier-based password-authenticated key exchange: new models and constructions. IACR Cryptology ePrint Archive 2013:833 (2013)
8. Bernstein, D.J., Hamburg, M., Krasnova, A., Lange, T.: Elligator: elliptic-curve points indistinguishable from uniform random strings. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & #38; Communications Security, CCS 2013, pp. 967–980. ACM, New York (2013)
9. Boyko, V., MacKenzie, P., Patel, S.: Provably secure password-authenticated key exchange using Diffie-Hellman. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 156–171. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45539-6_12
10. Camenisch, J., Casati, N., Gross, T., Shoup, V.: Credential authenticated identification and key exchange. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 255–276. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_14
11. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: IEEE Symposium on Foundations of Computer Science - FOCS 2001, pp. 136–145. IEEE (2001)
12. Canetti, R., Halevi, S., Katz, J., Lindell, Y., MacKenzie, P.: Universally composable password-based key exchange. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 404–421. Springer, Heidelberg (2005). https://doi.org/10.1007/11426639_24
13. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 186–194. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-47721-7_12
14. Gennaro, R., Lindell, Y.: A framework for password-based authenticated key exchange. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 524–543. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-39200-9_33

15. Gentry, C., MacKenzie, P., Ramzan, Z.: A method for making password-based key exchange resilient to server compromise. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 142–159. Springer, Heidelberg (2006). https://doi.org/10.1007/11818175_9
16. Jarecki, S., Krawczyk, H., Xu, J.: OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-computation Attacks. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10822, pp. 456–486. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78372-7_15
17. Jutla, C., Roy, A.: Relatively-sound NIZKs and password-based key-exchange. In: Fischlin, M., Buchmann, J., Manulis, M. (eds.) PKC 2012. LNCS, vol. 7293, pp. 485–503. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30057-8_29
18. Jutla, C.S., Roy, A.: Dual-system simulation-soundness with applications to UC-PAKE and more. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015. LNCS, vol. 9452, pp. 630–655. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48797-6_26
19. Jutla, C.S., Roy, A.: Smooth NIZK arguments with applications to asymmetric UC-PAKE and threshold-IBE. IACR Cryptology ePrint Archive 2016:233 (2016)
20. Katz, J., Ostrovsky, R., Yung, M.: Efficient password-authenticated key exchange using human-memorable passwords. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 475–494. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44987-6_29
21. Katz, J., Vaikuntanathan, V.: Round-optimal password-based authenticated key exchange. J. Cryptol. **26**(4), 714–743 (2013)
22. Kiefer, F., Manulis, M.: Zero-knowledge password policy checks and verifier-based PAKE. In: Kutyłowski, M., Vaidya, J. (eds.) ESORICS 2014. LNCS, vol. 8713, pp. 295–312. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11212-1_17
23. MacKenzie, P.: More efficient password-authenticated key exchange. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 361–377. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45353-9_27
24. MacKenzie, P., Patel, S., Swaminathan, R.: Password-authenticated key exchange based on RSA. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 599–613. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44448-3_46
25. National Institute of Standards and Technology, U.S. Fips pub 186-4: Digital Signature Standard (DSS), July 2013. https://csrc.nist.gov. Accessed 2013
26. Tibouchi, M., Kim, T.: Improved elliptic curve hashing and point representation. Des. Codes Cryptogr. **82**(1–2), 161–177 (2017)