Owl: An Augmented Password-Authenticated Key Exchange Scheme

Feng Hao[1], Samiran Bag[2], Liqun Chen[3], and Paul C. van Oorschot[4]

[1] University of Warwick, UK. feng.hao@warwick.ac.uk
[2] Alan Turing Institute, UK. samiran.bag@gmail.com
[3] University of Surrey, UK. liqun.chen@surrey.ac.uk
[4] Carleton University, Canada. paulv@scs.carleton.ca

Abstract. We present Owl, an augmented password-authenticated key exchange (PAKE) protocol that is both efficient and supported by security proofs. Owl is motivated by recognized limitations in SRP-6a and OPAQUE. SRP-6a is the only augmented PAKE that has enjoyed wide use in practice to date, but it lacks the support of formal security proofs, and does not support elliptic curve settings. OPAQUE was proposed in 2018 as a provably secure and efficient alternative to SRP-6a, and was chosen by the IETF in 2020 for standardization, but open issues leave it unclear whether OPAQUE will replace SRP-6a in practice. Owl is obtained by efficiently adapting J-PAKE to an asymmetric setting, providing additional security against server compromise yet with lower computation than J-PAKE. Our scheme is provably secure, efficient and agile in supporting implementations in diverse multiplicative groups and elliptic curve settings. Owl is the first solution that provides systematic advantages over SRP-6a in terms of security, computation, message sizes, and agility. Owl's agility across settings also contrasts ongoing issues related to how OPAQUE will instantiate a hash-to-curve operation in the elliptic curve setting (and what impact this will have on efficiency, security and forward compatibility with new elliptic curves in the future).

Keywords: Password authenticated key exchange, Augmented PAKE

1 Introduction

Password authenticated key exchange (PAKE) represents a major category of cryptographic protocols that allow two parties to establish a high-entropy session key based on a shared low-entropy secret (e.g., a memorable password) without requiring any public key infrastructure (PKI). This line of research started with the 1992 Bellovin-Merritt Encrypted Key Exchange (EKE), and has become a busy field. In contrast to using passwords for authentication in TLS, PAKE never discloses the plaintext password to the verifier during the authentication process, and hence is naturally resistant to phishing attacks. Furthermore, it is particularly useful in applications where a PKI is unavailable or untrusted.

There are two types of PAKE protocols in general: balanced and augmented (also resp. called symmetric and asymmetric PAKE). In the former, two parties share a common secret (e.g., a password or a hash of a password). In the latter, which is customized for a client-server setting, a client holds a password but the server stores only a one-way transformation of it. The idea is that reversing the transformation requires an offline search that proceeds by guessing through an enumeration of candidate passwords. Augmented PAKE improves security (vs. balanced PAKE) in case of *server compromise*: in a balanced PAKE, any plaintext credentials stolen from a server can be directly used to impersonate clients, but in an augmented PAKE, to recover plaintext passwords requires an offline guessing attack. This is an easy way to make attacks more expensive, without involving tamper-resistant hardware or multiple servers [25].

Augmented PAKE has received less attention to date than balanced PAKE. The additional security requirement makes it more complex to design. Several augmented PAKE protocols are available in the literature, but only SRP-6a [36] has been widely implemented in practice. (It is the latest version of SRP-3, following a series of revisions to patch weaknesses in the 1998 Secure Remote Password 3 protocol [38].) For example, SRP-6a is used in iCloud, 1Password and ProtonMail [18]. Security concerns [26] have been raised related to its heuristic design. The protocol also requires working over the whole range of a multiplicative group $\mathbb{Z}_p^*$ (where p is a safe prime, i.e., $p = 2q + 1$ with q also prime). This makes modular exponentiation relatively expensive—e.g., given a 3072-bit p, the exponent for modular exponentiation is 3072 bits, and one exponentiation is $3072/256 = 12$ times as costly as an exponentiation in 3072-bit DSA with 256-bit exponents. Finally, SRP-6a does not support elliptic curve implementations.

OPAQUE is an augmented PAKE scheme [26] asserted to have advantages over SRP-6a. For reasons explained in §2, it remains unclear whether OPAQUE will displace SRP-6a in practice. Finding an alternative more secure and efficient than SRP-6a has been an open problem to date. We address this herein by efficiently adapting J-PAKE [17] to an asymmetric setting, yielding a new scheme: Owl.[5] Compared with J-PAKE, Owl provides additional security against server compromise with even lower computation overall. To the best of our knowledge, Owl is the first protocol that shows systematic advantages over SRP-6a (see §5 for a comprehensive comparison). Our contributions are as follows.

1. We propose Owl, a novel augmented PAKE scheme. We do this by modifying J-PAKE, delivering extra security features yet with lower computation.
2. We formally prove the security of Owl under the Computational Diffie-Hellman (CDH) and Decision Diffie-Hellman (DDH) assumptions in a Universal Composability (UC) framework in the random oracle model.
3. We show that Owl is systematically better than SPR-6a in terms of security, computation, message sizes, and cryptographic agility in implementations.

[5] Owls have asymmetric ears with one higher than the other. This asymmetry helps owls pinpoint the source of a sound in darkness.

2 Motivation and background

Owl is motivated by limitations that have been observed in SRP-6a, and different issues that are now known with OPAQUE. This section provides the context.

To date, many more balanced than augmented PAKE schemes have been proposed. Among the many balanced PAKE schemes, a smaller number have been implemented in real-world applications, including SAE [19] in Wi-Fi Protected Access 3 (WPA3), J-PAKE [17] in IoT (as part of the Thread specification) and browser sync, PACE [7] in e-passports (third generation), and SPEKE [23] in Blackberry Messenger. For a state-of-the-art review of PAKE protocols and their real-world applications, see Hao and van Oorschot [18].

SRP-6a is the only augmented PAKE that has been widely used in practice to date. However, aside from concerns about its heuristic security, it has practical limitations including (1) costly modular exponentiation due to the use of long exponents, and (2) failure to support elliptic curve implementations.

OPAQUE is an augmented PAKE scheme from 2018 by Jarecki et al. [26]. An advantage promoted in its favor is so-called *pre-computation security*: if a server is compromised, an attacker cannot use pre-computed tables to speed up offline search. In contrast, for some augmented schemes, a single pre-computed table can be used thereafter to efficiently recover many passwords. Compared to SRP-6a, however, the advantage is less: if a server is compromised, while it is possible for an attacker to speed up password guessing using a pre-computed table, a *unique* table must be pre-computed per user (because of SRP-6a's salt), significantly increasing attack costs in terms of memory used and pre-computation time. We emphasize that pre-computation security does not itself stop offline attacks—in case of server compromise, OPAQUE passwords remain at risk and should be updated as soon as possible (as for other augmented PAKE protocols).

In 2020, the IETF conducted a PAKE competition. OPAQUE was selected as a winner in the augmented PAKE category [8], with its pre-computation security regarded as an advantage over some other schemes. OPAQUE also has formal security proofs (SRP-6a does not), and seems to be much more efficient than others (for caveats, see §5). However, it has three limitations, as now detailed.

1) OPAQUE relies on a *hash-to-curve* (H2C) function to map a password to a random prime-order generator on an elliptic curve (EC), as part of the Oblivious PRF (OPRF) construction. However, this H2C function was not instantiated in the original paper. In fact, researchers have been trying to define H2C since 2000 (as part of the IEEE P1363.2 standardization project [22]), but even today, standard constructions of H2C remain missing. Earlier constructions of H2C in IEEE 1363.2 worked with general elliptic curves and guaranteed the correctness in the output (a prime-order generator), but they turned out to be vulnerable against side-channel timing attacks [18]. (The IEEE 1362.2 standard was withdrawn in 2019.) When OPAQUE was selected by IETF in 2020, H2C remained uninstantiated [16]. Since then, attempts to fill this gap by defining custom H2C functions continue, within an Internet draft [20]. The IETF H2C constructions are motivated to guarantee constant-time operations, but unlike the earlier IEEE constructions, they work only with specifically selected elliptic

curves and do not guarantee the output is a prime-order generator [18]. Until the H2C functions are finally established (defined) and widely adopted, how to securely and efficiently instantiate H2C remains an open problem.

2) The critical dependence on H2C essentially leaves OPAQUE undefined in a multiplicative group (MODP) setting. During the IETF selection process, OPAQUE was specified only in an EC setting. As shown in §5, it is possible to instantiate OPAQUE in a MODP setting, e.g., by replacing H2C with an equivalent hash-to-group (H2G) function as used in SPEKE [23], but the performance advantages of OPAQUE then diminish significantly.

3) After OPAQUE was selected by IETF, it was discovered that the original protocol reveals information to passive observers about whether the password has been recently changed during a login session [16]. Although this revelation may at first appear minor, it can be of significant concern in practice. Recall that the main motivation for augmented PAKE is to address the threat of "server compromise". In principle, with the stolen credentials from a compromised server, the attacker is able to launch brute-force attacks to uncover plaintext passwords, but this is a time-consuming process. If a user diligently updates the password, her account on the server can remain secure. However, many users may be slow to update passwords. By passively monitoring the login sessions for all users, an attacker learns valuable information to identify those who have not changed the password and hence prioritizes the brute-force attack against those users. SPR-6a does not appear to have this problem.

These above issues point to the need for an augmented PAKE scheme that is efficient across both multiplicative and elliptic curve settings, is supported by security proofs, and does not reveal information about password changes.

3 Protocol specification of Owl

Here we provide a specification of the Owl protocol. Owl follows the design strategy of J-PAKE by adopting Schnorr zero-knowledge proofs to enforce each party to honestly follow the protocol specification, but unlike J-PAKE, works in an *asymmetric* setting and delivers additional security against server compromise. To ease comparison, we reuse J-PAKE notation where possible, including x_1, x_2, x_3 and x_4 for J-PAKE's four ephemeral private keys. In the original J-PAKE scheme, x_1 and x_3 are never used again after their public keys g^{x_1} and g^{x_3} (together with the Schnorr zero-knowledge proofs) are computed. This observation provides us with an insight to securely and efficiently modify J-PAKE in the new setting to achieve *more* security with *less* computation by leveraging the data storage that becomes available on the server, as will be detailed below.

3.1 Setup

We describe the protocol in a DSA-like group G (the specification works the same in an elliptic curve setting). Let p and q be two large primes such that

$q \mid p-1$. The protocol operates in the subgroup of $\mathbb{Z}_p^*$ of prime order q. Any non-identity element in this subgroup can serve as a generator, denoted g. Unless specified otherwise, all modular operations are performed with reference to the modulus p. Let t be a (low-entropy) user-specific secret, obtained by hashing the `username` and `password`: $t = H(\texttt{username}\|\texttt{password}) \bmod q$. Here $\|$ denotes concatenation. In a practical implementation, each concatenated item can be prepended with the item's byte length, to clearly separate items. We define $\pi = H(t) \bmod q$ as a shared secret to be used for authenticated key exchange and $T = g^t \bmod p$ as a password verifier to be stored on the server. In Owl, $\pi, \pi + q, \pi + 2q, \cdots$ are *equivalent* values in the exponent, hence the '$\bmod q$' in the π definition. As in J-PAKE, we require $\pi \neq 0 \bmod q$. As summarized in Fig. 1, the protocol comprises two main phases: initial registration and login (authenticated key exchange); for completeness, we also cover password update.

Zero-knowledge proof. Given a private key $x \in_R [0, q-1]$ and the corresponding public key $X = g^x \bmod p$, ZKP$\{x : g, X\}$ will denote a zero-knowledge proof (ZKP) for conveying knowledge of exponent x. When the context is clear, we shorten this to ZKP$\{x\}$. As a concrete instantiation, we use Schnorr non-interactive zero-knowledge (NIZK) proof [15], which is provably secure, revealing nothing beyond the truth of the statement: "the prover knows the exponent".

Specifically, to generate a zero-knowledge proof for ZKP$\{x : g, X\}$, the prover chooses a random secret $v \in_R [0, q-1]$, computes $V = g^v \bmod p$, and outputs (h, r) as the "proof", where $h = H(g\|V\|X\|\texttt{ProverID})$ and $r = v - x \cdot h \bmod q$. `ProverID` represents the prover's unique identity. $H(\cdot)$ is a secure one-way hash function.[6] Verification of the ZKP requires that the verifier check:

1) X has the correct prime order; and
2) $h \overset{?}{=} H(g\|g^r \cdot X^h\|X\|\texttt{ProverID})$, where h is the received value.

Note that in the original J-PAKE protocol, the Schnorr ZKP contains (V, r) while we use (h, r). The two are equivalent [15, §4] with the same computation cost, but the latter has a more compact size in a MODP setting. In Owl, if the ZKP verification fails, the receiver will reject the received data with a "ZKP verification failure" message. The status of the session remains intact.

3.2 Initial registration

To register an account on a server, a user computes $t = H(U\|w) \bmod q$. Here U denotes her `username`; w her (weak) `password`. She then computes $\pi = H(t) \bmod q$, $T = g^t \bmod p$, and sends (U, π, T) to the server through a secure channel (e.g., over TLS or out-of-band). The server chooses a random secret $x_3 \in_R [1, q-1]$, and computes a public key $X_3 = g^{x_3} \bmod p$ together with a zero-knowledge proof $\Pi_3 = \text{ZKP}\{x_3 : g, X_3\}$. The server stores $\{U : X_3, \Pi_3, \pi, T\}$ as the password verification file for U, and deletes x_3.

[6] The use of a secure one-way hash function is a common technique to transform an interactive ZKP into a non-interactive one based on the Fiat-Shamir heuristics [10]. As a result, the security proof for Owl is in the random oracle model.

Init registration		
User (U, w)		**Server** (S)
$t = H(U\|w) \bmod q$		$x_3 \in_R [0, q-1], X_3 = g^{x_3}$
$\pi = H(t) \bmod q$		$\Pi_3 = \text{ZKP}\{x_3\}$
$T = g^t \bmod p$	$\xrightarrow{U, \pi, T}$ (Secure channel)	Stores $\{U : X_3, \Pi_3, \pi, T\}$
Login		
User (U, w)		**Server** $(S, \{U : X_3, \Pi_3, \pi, T\})$
$t = H(U\|w) \bmod q$		
$\pi = H(t) \bmod q$		
$x_1 \in_R [0, q-1], X_1 = g^{x_1}$		
$\Pi_1 = \text{ZKP}\{x_1\}$		
$x_2 \in_R [1, q-1], X_2 = g^{x_2}$		
$\Pi_2 = \text{ZKP}\{x_2\}$	$\xrightarrow{U, X_1, X_2, \Pi_1, \Pi_2}$	Verify $\Pi_1, \Pi_2, X_2 \neq 1$
		$x_4 \in_R [1, q-1], X_4 = g^{x_4}$
		$\Pi_4 = \text{ZKP}\{x_4\}$
		$\beta = (X_1 X_2 X_3)^{x_4 \cdot \pi}$
Verify $\Pi_3, \Pi_4, \Pi_\beta, X_4 \neq 1$	$\xleftarrow{S, X_3, X_4, \Pi_3, \Pi_4, \beta, \Pi_\beta}$	$\Pi_\beta = \text{ZKP}\{x_4 \cdot \pi\}$
$\alpha = (X_1 X_3 X_4)^{x_2 \cdot \pi}$		
$\Pi_\alpha = \text{ZKP}\{x_2 \cdot \pi\}$		
$K = (\beta / X_4^{x_2 \cdot \pi})^{x_2}$		
$h = H(K\|\texttt{Transcript})$		
$r = x_1 - t \cdot h \bmod q$	$\xrightarrow{\alpha, \Pi_\alpha, r}$	Verify Π_α
		$K = (\alpha / X_2^{x_4 \cdot \pi})^{x_4}$
		$h = H(K\|\texttt{Transcript})$
		Verify $g^r \cdot T^h = X_1$
$k = H(K)$		$k = H(K)$
Password update		
User (U, w, w')		**Server** (S, $\{U : X_3, \Pi_3, \pi, T\}$)
$t' = H(U\|w') \bmod q$		
$\pi' = H(t') \bmod q$		
$T' = g^{t'} \bmod p$	$\xrightarrow{\pi', T'}$ (Secure PAKE channel)	Replaces (π, T) with (π', T')

Fig. 1. The Owl protocol. U is the user's identity, w her (weak) password.

We will use π as a shared secret to run a modified J-PAKE protocol by using the pre-computed X_3 and Π_3 values instead of freshly generating them for each login session as in the original J-PAKE protocol. This modification does not affect the security of the session key, as we will show in §4. We define π as a secret salted by a unique username (rather than $H(w)$). This is to ensure that upon server compromise if an attacker wishes to use a pre-computed table to launch an offline dictionary attack, he needs to build a *unique* pre-computed table for each user. In addition to modifying J-PAKE, we adopt a method proposed by

Hwang et al. to enable a client to securely prove the knowledge of t for $T = g^t$ based on a variant of Schnorr NIZK proof [21] with details below.

3.3 Login

The protocol runs between a client user (with unique identity U) and server (with identity S). The user initiates the communication. Both sides check: $U \neq S$. There are three flows in the login process to perform *authenticated key exchange*.

1. User $\rightarrow$ Server: U chooses random private keys $x_1 \in_R [0, q-1]$, $x_2 \in_R [1, q-1]$, computes corresponding public keys $X_1 = g^{x_1}$, $X_2 = g^{x_2} \bmod p$, and zero-knowledge proofs $\Pi_1 = \text{ZKP}\{x_1 : g, X_1\}$, $\Pi_2 = \text{ZKP}\{x_2 : g, X_2\}$ to prove knowledge of exponents x_1 and x_2. U sends to S: $(U, X_1, X_2, \Pi_1, \Pi_2)$.
2. Server $\rightarrow$ User: After verifying the received ZKPs and that $X_2 \neq 1 \bmod p$, S chooses a random private key $x_4 \in_R [1, q-1]$, computes $X_4 = g^{x_4} \bmod p$ and $\Pi_4 = \text{ZKP}\{x_4 : g, X_4\}$ to prove the knowledge of x_4, and computes $\beta = (X_1 X_2 X_3)^{\pi \cdot x_4}$ together with $\Pi_\beta = \text{ZKP}\{\pi \cdot x_4 : X_1 X_2 X_3, \beta\}$ to prove the knowledge of the exponent $\pi \cdot x_4$. S sends to U: $(S, X_3, X_4, \Pi_3, \Pi_4, \beta, \Pi_\beta)$.
3. User $\rightarrow$ Server: U verifies the received ZKPs and that $X_4 \neq 1 \bmod p$. U computes $\alpha = (X_1 X_3 X_4)^{x_2 \cdot \pi}$ together with $\Pi_\alpha = \text{ZKP}\{\mathrm{x}_2 \cdot \pi : \mathrm{X}_1\mathrm{X}_3\mathrm{X}_4, \alpha\}$ to prove knowledge of the exponent $x_2 \cdot \pi$. U then computes $K = (\beta / X_4^{x_2 \cdot \pi})^{x_2} = g^{(x_1+x_3)\cdot x_2 \cdot x_4 \cdot \pi}$ and $r = x_1 - t \cdot h \bmod q$ where $t = H(U \| w)$ and $h = H(K \| \texttt{Transcript})$. Our definition of $\texttt{Transcript}$ herein is a record of the items exchanged between U and S for them to compute a common session key. $\texttt{Transcript} = U \| X_1 \| X_2 \| \Pi_1 \| \Pi_2 \| S \| X_3 \| X_4 \| \Pi_3 \| \Pi_4 \| \beta \| \Pi_\beta \| \alpha \| \Pi_\alpha$. The computation of r is based on a method proposed by Hwang et al. [21] to prove the knowledge of t for g^t in a compiler; here, we use X_1 as a commitment (which is included in $\texttt{Transcript}$). Finally, U sends to S: (α, Π_α, r).

Session key computation. After the third flow, the server S first verifies the received Π_α, and then computes $K = (\alpha / X_2^{x_4 \cdot \pi})^{x_4} = g^{(x_1+x_3)\cdot x_2 \cdot x_4 \cdot \pi}$. S then computes $h = H(K \| \texttt{Transcript})$. S checks that $g^r \cdot T^h = X_1 \bmod p$; otherwise, it rejects the login with "authentication failure". If U and S have used the correct password credentials, they will derive a common session key (for simplicity, using a one-way hash H as a key derivation function): $k = H(K) = H(g^{(x_1+x_3)\cdot x_2 \cdot x_4 \cdot \pi})$.

Explicit key confirmation. After each party has computed the session key, they use it to encrypt messages (in an authenticated mode) for secure communication. The equality of session keys can be verified based on whether the receiver can decrypt ciphertexts successfully; this is called *implicit key confirmation.* Alternatively, explicit assurance (before secure communication) that both parties have computed the same session key can be achieved by adding an *explicit key confirmation* procedure. There are a number of ways to do this. One example is to use the method from J-PAKE RFC 8236 (itself based on NIST SP 800-56A). We define $K = g^{(x_1+x_3)\cdot x_2 \cdot x_4 \cdot \pi}$ as the raw keying material, and derive a key-confirmation key $k' = H(K \| \texttt{"KC"})$. In the third flow, the user can append a key confirmation string $M_U = \text{HMAC}(k', U\|S\|X_1\|X_2\|X_3\|X_4)$ using an HMAC algorithm with k' being the key. The server replies with another key confirmation string $M_S = \text{HMAC}(k', S\|U\|X_3\|X_4\|X_1\|X_2)$. This requires an extra flow. Note that although explicit key confirmation is generally recommended, it is not mandatory in IEEE 1363.2, ISO/IEC 11770-4 and NIST SP 800-56A.

We note that in Owl's third flow, the client-to-server authentication has been done based on verification against the stored password verifier. The explicit key confirmation string from the client to the server can be piggybacked in the same flow at a negligible cost. After receiving the explicit assurance of key confirmation from the client, the server may use the session key to send encrypted data in the next flow, and piggyback the key confirmation string in the same flow, hence adding a negligible cost for realizing explicit key confirmation.

3.4 Password update

After the successful authenticated key exchange process, both parties use the session key k to create a secure channel. Through this secure channel, the user can update the password by sending π', T' to the server as shown in Fig. 1. The server updates the password verifier file for U accordingly. One reason for a user to update her password is suspicion that the old password was compromised. Owl's forward secrecy property (§4.3) ensures that a passive attacker who knows the old password cannot learn the session key k, and hence cannot learn the new password. Although OPAQUE and SRP-6a do not discuss this explicitly, they can use a similar method to update the password.

However, OPAQUE's login process can reveal information about whether a password has been recently updated as discussed in §2. In OPAQUE, during the registration phase, the server saves a ciphertext $c = E_m(\ldots)$ using a password-derived encryption key $m = H(w, f(w)^k)$ where w is the password, $f(w)$ is a function that maps the password to a generator in the designated prime-order subgroup, and $k \in_R \mathbb{Z}_q$ is a random secret. We omit the content of the encryption as it is not relevant to the discussion here. The same ciphertext c is sent back to the user in every login session, but a passive attacker can observe that the c value will change once the password is updated. Addressing this issue requires modifying the OPAQUE protocol. By comparison, Owl and SRP-6a do not have this problem. In Owl, the values (X_3, Π_3) remain the same after the password is updated. SRP-6a saves a salt s and a verifier $v = g^{H(s,w)}$ during the registration phase. The salt s is sent to the user in every login session. Although not explicitly discussed in SRP-6a [36], it is possible to update only $v' = g^{H(s,w')}$ with a new password w' without changing s, hence preventing the problem as in OPAQUE.

4 Security analysis

This section analyzes the security of Owl based on a universally composable (UC) security model due to Gentry, MacKenzie and Ramzan [12], which was derived from the UC PAKE model of Canetti et al. [6] by extending it from a symmetric setting to an asymmetric one. Based on the model of Gentry et al., Hwang et al. [21] proposed a compiler to convert symmetric PAKE to asymmetric PAKE based on a variant of the Schnorr non-interactive zero-knowledge proof. We adopt the method of Hwang et al. as part of our protocol (more specifically, proving the knowledge of t for $T = g^t$; see Fig. 1). As Hwang et al.'s compiler has been proven secure in the UC model on the assumption that the session key from the symmetric PAKE is secure, we focus on analyzing the security of the session key derived from the modified J-PAKE protocol.

4.1 Security assumption

The security proof of Owl is based on the intractability of Multiple Decision Diffe-Hellman (MDDH) [29] and Square Computational Diffie-Hellman (SCDH) assumptions. We show that Decision Diffe-Hellman (DDH) and MDDH assumptions are equivalent. The SCDH and Computational Diffie-Hellman (CDH) assumptions are also equivalent, as shown by Bao et al. [2].

Assumption 1 (DDH): *For any PPT adversary $\mathcal{A}$, $Adv_{\mathcal{A}}^{DDH}(\lambda) \leq negl(\lambda)$, where,*

$$Adv_{\mathcal{A}}^{DDH}(\lambda) = \Big| Pr[\mathcal{A}(g, g^a, g^b, g^{ab}) = 1 \big| a, b \xleftarrow{\$} \mathbb{Z}_q] - Pr[\mathcal{A}(g, g^a, g^b, g^c) \big| a, b, c \xleftarrow{\$} \mathbb{Z}_q] \Big|$$

Assumption 2 (MDDH): *For $n \in poly(\lambda)$, any PPT adversary $\mathcal{A}$, the following two distributions are computationally indistinguishable.*

$$R_0 = \Big((g, g^a, g^{b_i}, g^{ab_i}) : i \in [1, n] \big| a, b_1, b_2, \ldots, b_n \xleftarrow{\$} \mathbb{Z}_q \Big)$$

$$R_1 = \Big((g, g^a, g^{b_i}, g^{c_i}) : i \in [1, n] \big| a, b_1, b_2, \ldots, b_n, c_1, c_2, \ldots, c_n \xleftarrow{\$} \mathbb{Z}_q \Big)$$

We define the advantage of $\mathcal{A}$ in distinguishing between R_0 and R_1 as,

$$Adv_{\mathcal{A}}^{MDDH}(\lambda) = \Big| Pr[\mathcal{A}(R_0) = 1] - Pr[\mathcal{A}(R_1) = 1] \Big|$$

Hence, $Adv_{\mathcal{A}}^{MDDH}(\lambda) \leq negl(\lambda)$, for a PPT adversary $\mathcal{A}$.

Lemma 1. *Assumption 1 and Assumption 2 are equivalent.*

Proof. ($\Rightarrow$): If there is an adversary $\mathcal{A}$ against Assumption 2, we could use it to construct another adversary $\mathcal{B}$ against Assumption 1. This has been proved in Lemma 4 by Kurosawa and Nojima [29]. ($\Leftarrow$): Let us assume that we have an adversary $\mathcal{B}$, against the DDH assumption. We use it to construct another adversary $\mathcal{A}$, against Assumption 2. $\mathcal{A}$ receives as input g, g^a, and b_i, $\Omega_{di} \in \{g^{a \cdot b_i}, R_i\}$, for $i \in [1, n]$, where $R_i \xleftarrow{\$} G$. It then invokes $\mathcal{B}$ with g, g^a, g^{b_k}, and Ω_{dk} for some $k \in [1, n]$. $\mathcal{B}$ has to check if $\Omega_{dk} = g^{a \cdot b_k}$ or a random element in G. If $\mathcal{B}$ is successful, so will be $\mathcal{A}$. Hence, the result holds.

Assumption 3 (CDH): *Given $x, y \xleftarrow{\$} \mathbb{Z}_q$, and $g \xleftarrow{\$} G$, define $CDH_g(g^x, g^y) = g^{xy}$. Consider the following security experiment $Exp_{\mathcal{A}}^{CDH}(\lambda)$. In this experiment, the challenger randomly chooses three elements g, $X = g^x$, $Y = g^y$ from G. Then the adversary $\mathcal{A}$ is invoked with these three as inputs. $\mathcal{A}$ has to compute $CDH_g(X, Y)$ from these parameters. The experiment is successful if $\mathcal{A}$ can compute $CDH_g(X, Y)$ correctly. The advantage of an adversary $\mathcal{A}$, against*

```
$Exp_{\mathcal{A}}^{CDH}(\lambda)$
$g \xleftarrow{\$} G$
$X \xleftarrow{\$} G$
$Y \xleftarrow{\$} G$
$C \leftarrow \mathcal{A}(g, X, Y)$
Return $C \stackrel{?}{=} CDH_g(X, Y)$
```

$Exp_{\mathcal{A}}^{CDH}(\lambda)$ is given by

$$Adv_{\mathcal{A}}^{CDH}(\lambda) = Pr[Exp_{\mathcal{A}}^{CDH}(\lambda) = 1]$$

As such, for any PPT adversary $\mathcal{A}$ $Adv_{\mathcal{A}}^{CDH}(\lambda) \leq negl(\lambda)$.

Assumption 4 (SCDH): *Given $x \xleftarrow{\$} \mathbb{Z}_q$, and $g \xleftarrow{\$} G$, define $SCDH_g(g^x) = g^{x^2}$. Consider the following security experiment $Exp_{\mathcal{A}}^{SCDH}(\lambda)$. In this experiment, the challenger randomly chooses two elements g and $X = g^x$ from G. Then the adversary $\mathcal{A}$ is invoked with these two as inputs. $\mathcal{A}$ has to compute $SCDH_g(X)$ from these parameters. The experiment is successful if $\mathcal{A}$ can compute $SCDH_g(X)$ correctly. The advantage of an adversary $\mathcal{A}$, against $Exp_{\mathcal{A}}^{SCDH}(\lambda)$*

```
$Exp_{\mathcal{A}}^{SCDH}(\lambda)$
$g \xleftarrow{\$} G$
$X \xleftarrow{\$} G$
$C \leftarrow \mathcal{A}(g, X)$
Return $C \stackrel{?}{=} SCDH_g(X)$
```

is given by

$$Adv_{\mathcal{A}}^{SCDH}(\lambda) = Pr[Exp_{\mathcal{A}}^{SCDH}(\lambda) = 1].$$

As such, for any PPT adversary $\mathcal{A}$ $Adv_{\mathcal{A}}^{SCDH}(\lambda) \leq negl(\lambda)$.

Lemma 2. *Assumption 3 and Assumption 4 are equivalent [2] .*

4.2 UC Secure augmented PAKE

In this section, we discuss the UC functionality $\mathcal{F}_{APAKE}$ of an augmented PAKE protocol based on the model of Gentry et al. [12]. Figure 2 provides the description of the ideal functionality $\mathcal{F}_{APAKE}$ of an augmented PAKE protocol. $\mathcal{F}_{APAKE}$ is parameterized by a security parameter λ. It interacts with an adversary $\mathcal{A}$ and other parties in the protocol via a set of the defined queries. Owl employs Schnorr ZKPs to enforce that each party honestly follows the protocol specification; only messages with valid ZKPs will be accepted and processed. Anyone (including the adversary) can verify the ZKPs. If a ZKP fails to be verified, the receiver aborts processing the received data *without* changing the state of the session (i.e., a malformed input does not affect the session). The session will be aborted only when the client's password and the server's password verification file do not match (i.e., an authentication failure), as already handled in Gentry et al.'s model.

The $(\texttt{StorePWFile}, sid, P_i, w)$ query from party P_j corresponds to the server storing a password verification file in its database, where *sid* is the index of the record $\langle FILE, P_i, P_j\rangle$. If there is already a record then the query is ignored. Else, the password verification file is recorded by the UC functionality $\mathcal{F}_{APAKE}$ in $file[sid]$. When $\mathcal{F}_{APAKE}$ receives a query ($\texttt{CltSession}$, sid, $ssid$, P_j, w) from the client P_i to start a session, $\mathcal{F}_{APAKE}$ checks if the $\texttt{CltSession}$ query is new or not. If there is no record of a previous $\texttt{CltSession}$ query, $\mathcal{F}_{APAKE}$ creates a record $\langle ssid, P_i, P_j, w\rangle$ where *ssid* represents a unique session ID and marks it `fresh`. If there is already a `fresh` record, the query is ignored. $\mathcal{F}_{APAKE}$ sends $(\texttt{CltSession}, sid, ssid, P_i, P_j)$ to the adversary $\mathcal{A}$. When $\mathcal{F}_{APAKE}$ receives a $(\texttt{SvrSession}, sid, ssid)$ query from the server P_j, it checks if there exists a password record $\langle FILE, P_i, P_j, w\rangle$. That is to say, the functionality checks if the client is already registered with the server or not. If there is no such record, the query is ignored. $\mathcal{F}_{APAKE}$ also checks if there exists a `fresh` $\texttt{CltSession}$ query for *ssid*, and ignores if not. Then $\mathcal{F}_{APAKE}$ sends $(\texttt{SvrSession}, sid, ssid, P_i, P_j)$ to the adversary $\mathcal{A}$. For every $\texttt{SvrSession}$ query that goes through, $\mathcal{F}_{APAKE}$ makes a session record $\langle ssid, P_j, P_i, w\rangle$ and marks it `fresh`.

Now, we discuss how $\mathcal{F}_{APAKE}$ deals with stealing password file queries. The 'Steal Password Query' is invoked by the adversary. In any balanced PAKE protocol, when the 'steal password file' query is invoked by the adversary, the ideal functionality sends the password directly to the adversary. However, in an augmented PAKE protocol, the adversary needs to do a brute-force search to find the password from the information obtained through the theft. The brute-force search is captured by the $\texttt{OfflineTestPwd}$ query. When the adversary makes this query with the correct password after compromising the server, the functionality returns 'correct guess'. This event happens when the brute-force search is done by the adversary on the correct password. If the password is wrong, the functionality returns 'wrong guess'. If the query is made before compromising the server, the functionality stores a record $(offline, w')$. This is done to model the construction of the pre-processing table by the adversary. If later the adversary compromises the server, and there is a matching entry in the preprocessing

Functionality $\mathcal{F}_{APAKE}$

Password storage and authentication sessions

1 Upon receiving a query $(\mathtt{STOREPWFILE}, sid, P_i, w)$ from party P_j:
- If there is a record $\langle FILE, P_i, P_j, w' \rangle$, then do nothing.
- Else create a `fresh` record $\langle FILE, P_i, P_j, w \rangle$ and mark it `uncompromised`.

2 Upon receiving a query $(\mathtt{CLTSESSION}, sid, ssid, P_j, w)$ from party P_i:
- Send $(\mathtt{CLTSESSION}, sid, ssid, P_i, P_j)$ to adversary $\mathcal{A}$, and if this is the first CLTSESSION query for $ssid$, store session record $\langle ssid, P_i, P_j, w \rangle$ and mark it `fresh`.

3 Upon receiving a query $(\mathtt{SVRSESSION}, sid, ssid)$ from P_j:
- If there is a password record $\langle FILE, P_i, P_j, w \rangle$, then send $(\mathtt{SVRSESSION}, sid, ssid, P_i, P_j)$ to $\mathcal{A}$, and if there exists a `fresh` CLTSESSION query for $ssid$, store session record $\langle ssid, P_j, P_i, w \rangle$, and mark it `fresh`.

Stealing Password Files

1 Upon receiving a query $(\mathtt{STEALPWFILE}, sid)$ from adversary $\mathcal{A}$:
- If there is no password data record, reply to $\mathcal{A}$ with "no password file". Otherwise, do the following. If the password data record $\langle FILE, P_i, P_j, w \rangle$ is marked `uncompromised`, mark it as `compromised`. If there is a tuple $(offline, w')$ stored with $w = w'$, send w to $\mathcal{A}$, otherwise reply to $\mathcal{A}$ with "password file stolen".

2 Upon receiving a query $(\mathtt{OFFLINETESTPWD}, sid, w')$ from adversary $\mathcal{A}$:
- If there is no password data record, or if there is a password data record $\langle FILE, P_i, P_j, w \rangle$ that is marked `uncompromised`, then store $(offline, w')$. Otherwise, do: If $w = w'$, reply to $\mathcal{A}$ with "correct guess". If $w \neq w'$, reply with "wrong guess".

Active session attacks

1 Upon receiving a query $(\mathtt{TESTPWD}, sid, ssid, P, w')$ from adversary $\mathcal{A}$:
- If there is a session record of the form $\langle ssid, P, P', w \rangle$ which is `fresh`, then do: If $w = w'$, mark the record `compromised` and reply to $\mathcal{A}$ with "correct guess". Otherwise, mark the record `interrupted` and reply with "wrong guess".

2 Upon receiving a query $(\mathtt{IMPERSONATE}, sid, ssid)$ from adversary $\mathcal{A}$:
- If there is a session record of the form $\langle ssid, P_i, P_j, w \rangle$ which is `fresh`, then do: If there is a password data record $\langle FILE, P_i, P_j, w \rangle$ that is marked `compromised`, mark the session record `compromised` and reply to $\mathcal{A}$ with "correct guess", else mark the session record `interrupted` and reply with "wrong guess".

Key Generation and Authentication

1 Upon receiving a query $(\mathtt{NEWKEY}, sid, ssid, P, k)$ from $\mathcal{A}$, where $|k| = \lambda$, if there is a record of the form $\langle ssid, P, P', w \rangle$ that is not marked `completed`, then:
- If this record is compromised, or either P or P' is corrupted, output $(sid, ssid, k)$ to P.
- If this record is `fresh`, there is a session record $\langle ssid, P', P, w' \rangle$, $w' = w$, a key k' was sent to P', and $\langle ssid, P', P, w \rangle$ was `fresh` at the time, then let $k'' = k'$, else pick a random key k'' of length λ. Output $(sid, ssid, k'')$ to P.
- In any other case, pick a random key $k'' \overset{\$}{\leftarrow} \{0,1\}^\lambda$, and output $(sid, ssid, k'')$ to P.
- Finally, mark the record $\langle ssid, P, P', w \rangle$ as `completed`.

2 Upon receiving a query $(\mathtt{TESTABORT}, sid, ssid, P)$ from $\mathcal{A}$: If there is a record of the form $\langle ssid, P, P', w \rangle$ that is not marked `completed`, then:
- If this record is `fresh`, there is a record $\langle ssid, P', P, w' \rangle$, and $w' = w$, let $b' = succ$.
- In any other case, let $b' = fail$.
- Send b' to $\mathcal{A}$. If $b' = fail$, send $(abort, sid, ssid)$ to P and mark $\langle ssid, P, P', w \rangle$ `completed`.

Fig. 2. UC functionality $\mathcal{F}_{APAKE}$ of augmented PAKE

Functionality $\mathcal{F}_{RO}$
Upon receiving a message $(Hash, sid, m)$ **from any party** P**:**
If there is a tuple (m, r) recorded, return r. Else, sample a random l-bit string $r \xleftarrow{\$} \{0,1\}^l$, store (m, r) and return r to P.

Fig. 3. The random oracle functionality.

table, the adversary gets the password by looking up the table rather than doing a brute-force search.

The `TESTPWD` query models the online password guessing attack. If the adversary picks the correct password, the attack is successful, and the adversary learns the password. The `IMPERSONATE` attack is successful if the adversary has made a `STEALPWFILE` before. If the adversary sends a `NEWKEY` query then the ideal functionality allows the adversary to set a key for a party that is either a corrupt party or whose partner is a corrupt party. If both the parties are honest and the session is not compromised then the functionality $\mathcal{F}_{APAKE}$ sends the same randomly generated key to both parties. In all other cases, the functionality assigns a random key to each party. The `TESTABORT` query is to let the session abort when the password authentication fails (i.e., the client's and the server's passwords do not match).

Random Oracle: We define the UC functionality of a random oracle in Figure 3. The oracle is queried with a message m. It returns a random string of a fixed bit length. If a query is repeated, the oracle returns whatever it had returned earlier. Our augmented PAKE protocol uses the random oracle functionality to compute 1) $t = H(U\|w)$; 2) $\pi = H(t)$; 3) the session key $k = H(K)$; 4) the challenge $h = H(K\|\texttt{Transcript})$ in Hwang et al.'s compiler [21]. To distinguish these, we use the argument $\langle sid, d\rangle$, where $d = 1, 2, 3, 4$ respectively.

Owl in the UC framework: Figure 4 describes the Owl protocol in the UC framework. During the registration phase, we follow the ideal functionality defined by Gentry et al. [12] to let the server process the password to derive a password verification file for storage. In practice, we can let the client compute this file instead, so the server has no access to the password (see §3 for more details). When P_i (or P_j) receives a `NEWKE` message, they check the correctness of the message format and parameters. If the format and parameters are correct, P_i and P_j compute a session key k and k' respectively. If P_i has used the password matching the verification file stored by the server P_j, we have $k = k'$; otherwise, k and k' are two random strings, $k \neq k'$ with an overwhelming probability.

The Simulator: In order to prove that our augmented PAKE protocol discussed in Figure 4 realizes the UC functionality presented in Figure 2, we need to introduce a simulator $\mathcal{SIM}$. In a simulated world, the adversary $\mathcal{A}$ interacts with the simulator which in turn interacts with the ideal functionality. Figure 5 depicts two scenarios: the real world and the simulated world. In both scenarios, there

UC Augmented PAKE protocol Owl

Setup: This protocol uses a random oracle functionality $\mathcal{F}_{RO}$.

Storing Password: Upon receiving $(\texttt{STOREPWFILE}, sid, P_i, w)$, P_j does the following:

- P_j sends $(Hash, \langle sid, 1\rangle, \langle P_i, w\rangle)$ to $\mathcal{F}_{RO}$ and receives the response t.
- P_j sends $(Hash, \langle sid, 2\rangle, t)$ to $\mathcal{F}_{RO}$ and receives the response π.
- P_j samples random $x_3 \overset{\$}{\leftarrow} \mathbb{Z}_q$, and stores $file[sid] = (X_3 = g^{x_3}, \Pi_3, \pi, T = g^t)$. Here, $\Pi_3 = ZKP[x_3 : g, g^{x_3}]$.

Protocol Steps:

1 When P_i receives $(\texttt{CLTSESSION}, sid, ssid, P_j, w)$, she sends $(Hash, \langle sid, 1\rangle, \langle P_i, w\rangle)$ to $\mathcal{F}_{RO}$ and receives the response t. She sends $(Hash, \langle sid, 2\rangle, t)$ to $\mathcal{F}_{RO}$ and receives the response π. P_i selects $x_1 \overset{\$}{\leftarrow} \mathbb{Z}_q$, $x_2 \overset{\$}{\leftarrow} \mathbb{Z}_q^*$, and computes $X_1 = g^{x_1}$, $X_2 = g^{x_2}$, $\Pi_1 = ZKP[x_1 : g, g^{x_1}]$, and $\Pi_2 = ZKP[x_2 : g, g^{x_2}]$. P_i sends to P_j $(\texttt{FlowOne}, sid, ssid, D)$, where $D = \langle X_1, X_2, \Pi_1, \Pi_2\rangle$.

2 When P_j receives $(\texttt{SVRSESSION}, sid, ssid, P_i, D)$, she obtains (X_3, Π_3, π, T) from the tuple stored in $file[sid]$ (aborting if this value is not properly defined). She then parses D and aborts if this parsing fails. P_j selects $x_4 \overset{\$}{\leftarrow} \mathbb{Z}_q^*$, and computes $X_4 = g^{x_4}$, $\beta = (X_1X_2X_3)^{x_4 \cdot \pi}$, $\Pi_4 = ZKP[x_4 : g, g^{x_4}]$ and $\Pi_\beta = ZKP[x_4 \cdot \pi : X_1X_2X_3, \beta]$. P_j sends to P_i $(\texttt{FlowTwo}, sid, ssid, E)$, where $E = \langle X_3, X_4, \beta, \Pi_3, \Pi_4, \Pi_\beta\rangle$.

3 When P_i receives $(\texttt{NEWKEY}, sid, ssid, P_j, E)$, she parses E and aborts if the parsing fails. P_i computes $\alpha = (X_1X_3X_4)^{x_2 \cdot \pi}$ and $\Pi_\alpha = ZKP[x_2 \cdot \pi : X_1X_3X_4, \alpha]$. P_i computes $K = (\beta / X_4^{x_2 \pi})^{x_2}$ and sends $(Hash, \langle sid, 4\rangle, \langle K, \texttt{Transcript}\rangle)$ to $\mathcal{F}_{RO}$ and obtains a response h. P_i computes $r = x_1 - t \cdot h \bmod q$ and sends to P_j $(\texttt{FlowThree}, sid, ssid, F)$, where $F = \langle \alpha, \Pi_\alpha, r\rangle$. Finally, P_i sends $(Hash, \langle sid, 3\rangle, K)$ to $\mathcal{F}_{RO}$ to obtain a session key k. P_i outputs $(sid, ssid, k)$ and terminates the session.

4 When P_j receives $(\texttt{NEWKEY}, sid, ssid, P_i, F)$, she parses F and aborts if the parsing fails. P_j computes $K' = (\alpha / X_2^{x_4 \cdot \pi})^{x_4}$, and sends $(Hash, \langle sid, 4\rangle, \langle K', \texttt{Transcript}\rangle)$ to $\mathcal{F}_{RO}$ to obtain h'. P_j verifies that $g^r \cdot T^{h'} = X_1$, and aborts the session if the verification fails. Finally, P_j sends $(Hash, \langle sid, 3\rangle, K')$ to $\mathcal{F}_{RO}$ to obtain a session key k'. P_j outputs $(sid, ssid, k')$ and terminates the session.

Stealing Password File: When P_j (who is a server) receives a message $(\texttt{STEALPWFILE}, sid)$, from the adversary $\mathcal{A}$, if $file[sid]$ is defined, P_j sends it to $\mathcal{A}$.

Fig. 4. The Owl protocol in the UC framework

is an environment $\mathcal{Z}$ that sets the passwords for the user and interacts with the dummy adversary $\mathcal{A}$. In the real world, $\mathcal{A}$ and the users interact with the Owl protocol. In the simulated world, the users interact with the ideal functionality, whereas $\mathcal{A}$ interacts with the simulator that in turn interacts with the ideal functionality. We will show that the view of $\mathcal{Z}$ in the two worlds is indistinguishable.

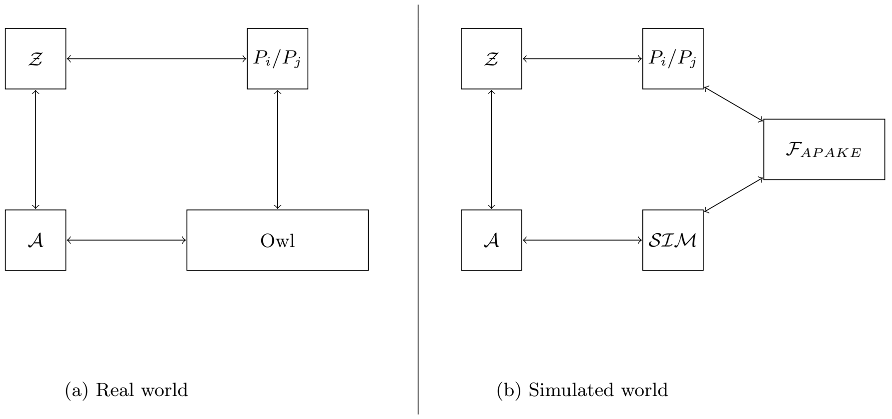

Fig. 5. View of environment for real world and simulated world.

The environment $\mathcal{Z}$ chooses an arbitrary password w from a dictionary $\mathcal{D}$ and performs the password registration by sending the StorePwdFile query to P_j, creating a record $\langle FILE, P_i, P_j, ss\rangle$, where ss represents the server storage, $ss = (X_3, \Pi_3, \pi, T)$. We emphasize that passwords chosen from the dictionary do not have to follow a uniform distribution; in fact, they can be of any distribution. The authentication in Owl is entirely based on the equality of the two (low-entropy) secret values chosen by both sides. In case of server compromise, the ss file will be revealed to $\mathcal{A}$ through the StealPwFile query. $\mathcal{SIM}$ must come up with a guessed password w' that matches $g^{H(U\|w')} = T$. $\mathcal{SIM}$ checks if the guessed password is correct through the OfflineTestPwd query.

When $\mathcal{Z}$ sends a client session initiation request to P_i, P_i chooses a password w', and sends a query (CltSession, $sid, ssid, P_i, P_j, w'$) to $\mathcal{F}_{APAKE}$, which then sends (CltSession, $sid, ssid, P_i, P_j$) to the simulator. $\mathcal{SIM}$ first checks if the session ID $ssid$ is `fresh`. If $ssid$ was used before, $\mathcal{SIM}$ ignores the request. If this $ssid$ is new, $\mathcal{SIM}$ creates a record $\langle ssid, P_i, P_j, \cdot\rangle$, and marks it `fresh`. $\mathcal{SIM}$ then chooses $x_1 \xleftarrow{\$} \mathbb{Z}_q$, $x_2 \xleftarrow{\$} \mathbb{Z}_q^*$, and generates NIZK proofs $\Pi_1 = ZKP[x_1 : g, g^{x_1}]$, and $\Pi_2 = ZKP[x_2 : g, g^{x_2}]$. $\mathcal{SIM}$ sends (`FlowOne`, $sid, ssid, D$) to $\mathcal{A}$, where $D = \langle g^{x_1}, g^{x_2}, \Pi_1, \Pi_2\rangle$.

When the adversary $\mathcal{A}$ sends a SvrSession request with $(id, ssid, P_i, P_j, D)$, $\mathcal{SIM}$ checks if there is a record $\langle FILE, P_i, P_j, ss\rangle$ and aborts if there is not (i.e., the client is not registered). If there is such a record, it retrieves $ss = (X_3, \Pi_3, \pi, T)$. $\mathcal{SIM}$ also checks if $\langle ssid, P_i, P_j, \cdot\rangle$ is `fresh`, and aborts if it is not.

$\mathcal{SIM}$ parses $D = \langle g^{x_1}, g^{x_2}, \Pi_1, \Pi_2 \rangle$ and aborts if any step in the parsing (which includes the verification of Π_1 and Π_2) fails. If the parsing is successful, it creates a record $\langle ssid, P_j, P_i, \cdot \rangle$ and marks it `fresh`. $\mathcal{SIM}$ selects $x_4 \xleftarrow{\$} \mathbb{Z}_q^*$ and generates a NIZK proof Π_4. $\mathcal{SIM}$ computes $\beta = (X_1 X_2 X_3)^{x_4 \cdot \pi}$ and generates a NIZK proof Π_β to prove the well-formedness of β. $\mathcal{SIM}$ sends a $(\texttt{FlowTwo}, sid, ssid, E)$ message to $\mathcal{A}$ where $E = \langle g^{x_3}, g^{x_4}, \beta, \Pi_3, \Pi_4, \Pi_\beta \rangle$.

When the adversary $\mathcal{A}$ sends a NEWKEY query, specifying $(sid, ssid, P_i, E)$, $\mathcal{SIM}$ first checks if the record $\langle ssid, P_i, P_j, \cdot \rangle$ is `fresh`, and aborts if it is not. $\mathcal{SIM}$ then parses $E = \langle g^{x_3}, g^{x_4}, \beta, \Pi_3, \Pi_4, \Pi_\beta \rangle$ and aborts if any step in the parsing (which includes verification of Π_3, Π_4 and Π_β) fails. If the parsing is successful, $\mathcal{SIM}$ extracts the NIZK proof Π_β to find the value of $x_4 \cdot \pi'$ and uses NIZK extraction to find x_4 from Π_4. $\mathcal{SIM}$ computes $x_4 \cdot \pi' / x_4 = \pi'$. $\mathcal{SIM}$ retrieves π from ss. If $\pi = \pi'$, $\mathcal{SIM}$ computes $K = (\beta / X_4^{x_2 \cdot \pi})^{x_2}$, sends a query $(Hash, \langle sid, 3 \rangle, K)$ to $\mathcal{F}_{RO}$ and uses the response k as the session key. If $\pi \neq \pi'$, $\mathcal{SIM}$ samples k randomly from $\{0,1\}^\lambda$. $\mathcal{SIM}$ outputs $(sid, ssid, k)$ to P_i. It updates the last item of the record $\langle ssid, P_i, P_j, \cdot \rangle$ with k and marks the record `completed`. $\mathcal{SIM}$ computes $\alpha = (X_1 X_3 X_4)^{x_2 \cdot \pi}$ and generates a NIZK proof Π_α to prove the well-formedness of α. Furthermore, $\mathcal{SIM}$ retrieves the SVRSESSION query and extracts the NIZK Π_1 to obtain x_1. $\mathcal{SIM}$ does not know $t = H(U \| w)$ without knowing the password. $\mathcal{SIM}$ must send an OFFLINETESTPWD query to $\mathcal{F}_{\mathcal{APAKE}}$ with a guessed password w'. If the response is "correct guess", $\mathcal{SIM}$ obtains $w = w'$ and computes $t = H(U \| w')$. $\mathcal{SIM}$ sends a query $(Hash, \langle sid, 4 \rangle, \langle K, \texttt{Transcript} \rangle)$ to $\mathcal{F}_{RO}$ to obtain a response h, and computes $r = x_1 - h \cdot t$. Finally, $\mathcal{SIM}$ sends $(\texttt{FlowThree}, sid, ssid, F)$ message to $\mathcal{A}$ where $F = \langle \alpha, \Pi_\alpha, r \rangle$.

When the adversary $\mathcal{A}$ sends a NEWKEY query, specifying $(sid, ssid, P_j, F)$, $\mathcal{SIM}$ first checks if the record $\langle ssid, P_j, P_i, \cdot \rangle$ is `fresh`, and aborts if it is not. $\mathcal{SIM}$ then parses $F = \langle \alpha, \Pi_\alpha, r \rangle$ and aborts if any step in the parsing (which includes verification of Π_α) fails. If the parsing is successful, $\mathcal{SIM}$ uses NIZK proof extraction to find the value of $x_2 \cdot \pi'$ from Π_α and the value of x_2 from Π_2. $\mathcal{SIM}$ computes $x_2 \cdot \pi' / x_2 = \pi'$. $\mathcal{SIM}$ retrieves the server storage ss from the record $\langle FILE, P_i, P_j, ss \rangle$, and parses $ss = (X_3, \Pi_3, \pi, T)$. If $\pi = \pi'$, $\mathcal{SIM}$ computes $K = (\alpha / X_2^{x_4 \cdot \pi})^{x_4}$. $\mathcal{SIM}$ sends a query $(Hash, \langle sid, 4 \rangle, \langle K, \texttt{Transcript} \rangle)$ to $\mathcal{F}_{RO}$ to obtain a response h, and aborts the session if $X_1 \neq g^r \cdot T^h$. $\mathcal{SIM}$ sends a query $(Hash, \langle sid, 3 \rangle, K)$ to $\mathcal{F}_{RO}$ and uses the response k as the session key. If $\pi' \neq \pi$, $\mathcal{SIM}$ samples k randomly from $\{0,1\}^\lambda$. $\mathcal{SIM}$ outputs $(sid, ssid, k)$ to P_j. It updates the last item of the record $\langle ssid, P_j, P_i, \cdot \rangle$ with k and marks the record `completed`.

4.3 Session key indistinguishability

We first show that the session key in Owl is indistinguishable from random under three scenarios: 1) when an active adversary impersonates the server without having the password verification file; 2) when an active adversary impersonates the client without knowing the password; 3) when a passive adversary knows the password. These results are needed for proving the main theorem later.

Impersonating the server We define a security experiment $Exp_{\mathcal{A}}^{RNDKey}(\lambda)$ to model the adversary impersonating the server. We use this experiment to show that the simulator $\mathcal{SIM}$ can replace the session key with the hash of a random element of G, and it will not be possible for the adversary to distinguish if the key it has received was the actual key or a randomly chosen key when the adversary had chosen an incorrect password. If the passwords used by the two parties do not match, both sides end up calculating different keys. In this attack scenario, the simulator responds to the queries of the attacker impersonating the server to a legitimate client without knowing the password verification file. The simulator follows the protocol specifications and sends appropriate information to the attacker against all its queries. In the end, the simulator calculates the session key. The simulator then flips a coin and returns either the actual session key or a random one depending upon the outcome of the coin tossing. We show that the attacker will not be able to distinguish between the two keys if the password chosen by the attacker w' is not the one (w) used by the simulator. So, if the session key is compromised, the attacker will learn nothing about the actual password used by the simulator.

In the security experiment $Exp_{\mathcal{A}}^{RNDKey}(\lambda)$, we define g as a random generator of the mathematical group G. $\mathcal{D}$ is a dictionary of passwords. $|\mathcal{D}| \in poly(\lambda)$. H is a secure hash function (modelled as $\mathcal{F}_{RO}$). The adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2)$ is a three stage adversary. Here, $CDH_g(A, B)$ denotes the Computational Diffie-Hellman of two elements A, and B with respect to g. Also $SCDH_g(A)$ denotes the square Computational Diffie-Hellman of an element A with respect to g. First, the challenger generates the public parameters. Then she chooses a generator g from G, and a password w from $\mathcal{D}$. The challenger chooses $X_1, X_2 \xleftarrow{\$} G$, and invokes $\mathcal{A}_0$ with the specific parameters as shown in the experiment. $\mathcal{A}_0$ outputs $x_2, x_4 \in \mathbb{Z}_q^*$. Then the challenger computes R, and invokes $\mathcal{A}_1$. $\mathcal{A}_1$ outputs a password guess $w' \in \mathcal{D}$ and obtains $\pi' = H(H(U\|w'))$. Then the challenger calculates a raw key K_0, and randomly samples K_1 from G. The challenger randomly chooses one of K_0 and K_1, and invokes $\mathcal{A}_2$ with its hash value. $\mathcal{A}_2$ has to identify whether $H(K_0)$ or $H(K_1)$ was passed to her as the input. The experiment is successful if $\mathcal{A}_2$ can identify the correct challenge.

Note that if $w = w'$ (or $\pi = \pi'$), K_0 will be equal to $CDH_g(X_1, X_2)^{x_4\pi} * X_2^{x_3x_4\pi}$. As such, $\mathcal{A}_2$ can distinguish between $H(K_0)$ and $H(K_1)$ easily. So, we define the advantage of $\mathcal{A}$ as $\left|Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1|\pi \neq \pi'] - \frac{1}{2}\right|$. Now, $Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1] = Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1|\pi = \pi'] * Pr[\pi = \pi'] + Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1|\pi \neq \pi'] * Pr[\pi \neq \pi']$. If $\pi = \pi'$, $\mathcal{A}_2$ can easily win the experiment. Therefore, $Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1|\pi = \pi'] = 1$. So, $Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1] = Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1|\pi \neq \pi'] * (1 - Pr[\pi = \pi']) + Pr[\pi = \pi'] = Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1|\pi \neq \pi'](1 - \frac{1}{|\mathcal{D}|}) + \frac{1}{|\mathcal{D}|}$. Now, $\mathcal{A}_2$ can always win with $\frac{1}{2}$ probability by making a random guess. Therefore, $Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1|\pi \neq \pi'] \geq \frac{1}{2}$. Let us assume that $Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1|\pi \neq \pi'] = \frac{1}{2} + X$, where X is used to define the advantage of $\mathcal{A}$ against $Exp_{\mathcal{A}}^{RNDKey}(\lambda)$. So, $Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1] = (X + \frac{1}{2})(1 - \frac{1}{|\mathcal{D}|}) + \frac{1}{|\mathcal{D}|}$. Therefore, $X = \frac{|\mathcal{D}|}{|\mathcal{D}|-1} * (Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) =$

$1] - \frac{1}{|\mathcal{D}|}) - \frac{1}{2} = \frac{|\mathcal{D}|}{|\mathcal{D}|-1} * (Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1] - \frac{1}{2|\mathcal{D}|} - \frac{1}{2})$. Since, $|\mathcal{D}|$ is $poly(\lambda)$, $\frac{|\mathcal{D}|}{|\mathcal{D}|-1}$ is a little higher than 1. So, we can eliminate the $\frac{|\mathcal{D}|}{|\mathcal{D}|-1}$ part in the above expression. Therefore, we define the advantage of an adversary $\mathcal{A}$ against the security experiment $Exp_{\mathcal{A}}^{RNDKey}(\lambda)$ as below:

$$Adv_{\mathcal{A}}^{RNDKey}(\lambda) = \left| Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1] - \frac{1}{2|\mathcal{D}|} - \frac{1}{2} \right|$$

```
$Exp_{\mathcal{A}}^{RNDKey}(\lambda)$
$(G, \mathcal{D}, H) \leftarrow Setup(1^\lambda)$
$g \xleftarrow{\$} G$
$w \leftarrow \mathcal{D}, \pi = H(H(U\|w))$
$X_1, X_2 \xleftarrow{\$} G$
$(x_3, x_4, state_0) \leftarrow \mathcal{A}_0(g, G, \mathcal{D}, X_1, X_2)$
$R \leftarrow (CDH_g(X_1, X_2) * X_2^{x_3+x_4})^\pi$
$(w', state_1) \leftarrow \mathcal{A}_1(state_0, R)$
$\pi' = H(H(U\|w'))$
$K_0 \leftarrow CDH_g(X_1, X_2)^{x_4 \cdot \pi'} * X_2^{x_3 \cdot x_4 \cdot \pi'} * SCDH_g(X_2)^{x_4 \cdot (\pi' - \pi)}$
$K_1 \xleftarrow{\$} G$
$d \xleftarrow{\$} \{0, 1\}$
$\Omega \leftarrow H(K_d)$
$d' \leftarrow \mathcal{A}_2(state_1, \Omega)$
Return $(d \overset{?}{=} d')$
```

Lemma 3. *Under the SCDH and DDH assumptions with access to $\mathcal{F}_{RO}$, for any PPT adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2)$, $Adv_{\mathcal{A}}^{RNDKey}(\lambda) \leq negl(\lambda)$.*

Proof. We show that if there exists an adversary $\mathcal{A}$ against the security experiment $Adv_{\mathcal{A}}^{RNDKey}$, we can use it to construct another adversary $\mathcal{B}$, against the security experiment $Exp_{\mathcal{A}}^{SCDH}(\lambda)$. $\mathcal{B}$ receives as input $A \xleftarrow{\$} G$. Its aim is to compute $SCDH_g(A)$. It selects random $x_1 \xleftarrow{\$} \mathbb{Z}_q$, and assigns $X_1 = g^{x_1}$, and $X_2 = A$. It invokes $\mathcal{A}_0$ and receives x_3, and x_4. Then $\mathcal{B}$ computes $R = X_2^{(x_1+x_3+x_4)\pi}$. $\mathcal{B}$ invokes $\mathcal{A}_1$ and receives w', thus $\pi' = H(H(U\|w'))$. If $\pi' = \pi$, $\mathcal{B}$ aborts and outputs a random element from G. Else, $\mathcal{B}$ samples a random string from $\{0,1\}^\lambda$, and assigns this to Ω. When $\mathcal{A}_0, \mathcal{A}_1$ or $\mathcal{A}_2$ makes an oracle query to H, $\mathcal{B}$ answers them. For each such query, $\mathcal{B}$ samples a random string and returns it. If a query is repeated, $\mathcal{B}$ returns the same string it had returned earlier. $\mathcal{B}$ keeps a log of all query-response pairs. After $\mathcal{A}_2$ has returned, $\mathcal{B}$ checks all the queries made by $\mathcal{A}_1$ or $\mathcal{A}_2$. It randomly selects one query-response pair $\langle K_0, k_0 \rangle$ where $K_0 = \left((X_1 X_2 X_3)^{x_4 \cdot \pi'} / X_2^{x_4 \cdot \pi} \right)^{x_2} = CDH_g(X_1, X_2)^{x_4 \cdot \pi'} * X_2^{x_3 \cdot x_4 \cdot \pi'} * SCDH_g(X_2)^{x_4 \cdot (\pi' - \pi)}$ represents the raw keying material computed by $\mathcal{A}$ and k_0 the session key. $\mathcal{B}$ computes $C = (K_0/(CDH_g(X_1, X_2)^{x_4\pi'} * X_2^{x_3 * x_4\pi'}))^{1/(x_4(\pi'-\pi))} = SCDH_g(X_2)$ and outputs C. Let us now calculate the success probability of $\mathcal{B}$. $\mathcal{B}$ wins if the following events happen together:

- $\pi \neq \pi'$
- $\mathcal{A}$ queries $H(K_0)$.

Since H is modelled as a random oracle $\mathcal{F}_{RO}$, $\mathcal{A}_2$ cannot identify d without querying $H(K_0)$. Thus, the probability that $\mathcal{A}$ queries $H(K_0)$ is at least equal to $Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1] - \frac{1}{2}$. Now, the probability that $H(K_0)$ will be queried and K_0 will have the term $SCDH_g(X_2)$ as a factor is $Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1] - \frac{Pr[\pi=\pi']}{2} - \frac{1}{2}$. First, we calculate the value of $Pr[\pi = \pi']$. If $\mathcal{A}$ can guess the value of π from R, then the probability of this event happening could be as high as 1. However, the probability that $\mathcal{A}$ can guess the value of π from R is bounded by $Adv_{\mathcal{A}}^{DDH}(\lambda)$ (i.e., distinguishing $CDH_g(X_1, X_2)$ from random). Thus, $Pr[\pi = \pi'] \leq \frac{1}{|\mathcal{D}|} + Adv_{\mathcal{A}}^{DDH}(\lambda)$. Hence, the probability that $\mathcal{A}$ will query K_0 is at least $Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1] - \frac{1}{2|\mathcal{D}|} - \frac{1}{2}Adv_{\mathcal{A}}^{DDH}(\lambda) - \frac{1}{2}$. Now, $\mathcal{B}$ randomly picks a query and computes $SCDH_g(A)$ on the basis of that. So if there are Q queries to H, the probability that $\mathcal{B}$ will pick the correct one is $1/Q$. Thus, $Adv_{\mathcal{B}}^{SCDH}(\lambda) \geq (1/Q) * (Pr[Exp_{\mathcal{A}}^{RNDKey}(\lambda) = 1] - \frac{1}{2|\mathcal{D}|} - \frac{1}{2}Adv_{\mathcal{A}}^{DDH}(\lambda) - \frac{1}{2}) = (1/Q)(Adv_{\mathcal{A}}^{RNDKey}(\lambda) - \frac{1}{2}Adv_{\mathcal{A}}^{DDH}(\lambda))$. Hence, $Adv_{\mathcal{A}}^{RNDKey}(\lambda) \leq Q * Adv_{\mathcal{B}}^{SCDH}(\lambda) + \frac{1}{2} * Adv_{\mathcal{A}}^{DDH}(\lambda)$.

Impersonating the client We consider a security experiment $Exp_{\mathcal{A}}^{RNDKey1}(\lambda)$, which emulates the event when the attacker tries to impersonate the user to a legitimate server without knowing the password. In this experiment, we introduce an oracle $\mathcal{O}$ that can be queried by the adversary. This oracle models the event that the adversary can exploit the fact the server uses the same value of X_3 across all the executions of the protocol. So, the adversary can try to impersonate the client and execute the protocol with the server multiple times and can receive `FlowTwo` messages from the server for different values of X_4, but a single value of X_3. In the end, the goal of the adversary is to distinguish between the session key computed by an honest instance and a random string. Similar to the case of $Exp_{\mathcal{A}}^{RNDKey}(\lambda)$ (which models an attacker impersonating the server), the advantage of the adversary $\mathcal{A}$, against the security experiment $Exp_{\mathcal{A}}^{RNDKey1}(\lambda)$ (which models an attacker impersonating the user) is defined as

$$Adv_{\mathcal{A}}^{RNDKey1}(\lambda) = \left| Pr[Exp_{\mathcal{A}}^{RNDKey1}(\lambda) = 1] - \frac{1}{2|\mathcal{D}|} - \frac{1}{2} \right|$$

Lemma 4. *Under the SCDH and MDDH assumptions with access to $\mathcal{F}_{RO}$, for any PPT adversary $\mathcal{A}$, $Adv_{\mathcal{A}}^{RNDKey1}(\lambda) \leq negl(\lambda)$.*

Proof. We can show that if there exists an adversary $\mathcal{A}$ against the security experiment $Exp_{\mathcal{A}}^{RNDKey1}(\lambda)$, we can use it in the construction of another adversary

$\mathcal{B}$ again the experiment $Exp_{\mathcal{A}}^{SCDH}(\lambda)$. The proof is almost the same as the proof of Lemma 3. The only difference is that because of the use of a fixed X_3 value across the sessions, we use Assumption 2 instead of Assumption 1. All other arguments are the same. Thus, we substitute the DDH assumption of Lemma 3 with Assumption 2, and have $Adv_{\mathcal{A}}^{RNDKey1}(\lambda) \leq Q * Adv_{\mathcal{A}}^{MDDH}(\lambda) + \frac{1}{2} * Adv_{\mathcal{A}}^{SCDH}(\lambda)$. Hence, the result follows.

```
$Exp_{\mathcal{A}}^{RNDKey1}(\lambda)$
$(G, \mathcal{D}, H) \leftarrow Setup(1^\lambda)$
$g \overset{\$}{\leftarrow} G$
$w \leftarrow \mathcal{D}, \pi = H(H(U \| w))$
$X_3, X_4 \overset{\$}{\leftarrow} G$
$d \overset{\$}{\leftarrow} \{0, 1\}$
$(x_1, x_2) \leftarrow \mathcal{A}^{\mathcal{O}}(g, G, D, X_3, X_4)$
$R \leftarrow (CDH_g(X_3, X_4) * X_4{}^{x_1+x_2})^\pi$
$w' \leftarrow \mathcal{A}(R)$
$\pi' = H(H(U \| w'))$
$B_0 \leftarrow CDH_g(X_3, X_4)^{x_2 \cdot \pi'} * X_4^{x_1 \cdot x_2 \cdot \pi'} * SCDH_g(X_4)^{x_2(\pi' - \pi)}$
$B_1 \overset{\$}{\leftarrow} G$
$d' \leftarrow \mathcal{A}(H(B_d))$
return $d \overset{?}{=} d'$
```

```
$\mathcal{O}()$
$X \overset{\$}{\leftarrow} G$
$(x_1, x_2) \leftarrow \mathcal{A}(X)$
$R \leftarrow (CDH_g(X_3, X) * X^{x_1+x_2})^\pi$
Return $R$
```

Session key exposure The following experiment $Exp_{\mathcal{A}}^{KeyExp}(\lambda)$ models the event where the attacker gets hold of an actual session key after the key is derived. We will show that the actual session key is indistinguishable from a random key in the view of the attacker even with the knowledge of the password (i.e., forward secrecy [17]). This can happen if $Adv_{\mathcal{A}}^{KeyExp}(\lambda)$ is negligible, where

$$Adv_{\mathcal{A}}^{KeyExp}(\lambda) = \left| Pr[Exp_{\mathcal{A}}^{KeyExp}(\lambda) = 1] - \frac{1}{2} \right|.$$

Lemma 5. *Under the DDH and SCDH assumptions with access to $\mathcal{F}_{RO}$, for any PPT adversary $\mathcal{A}$, $Adv_{\mathcal{A}}^{KeyExp}(\lambda) \leq negl(\lambda)$.*

Proof. We show that if there exists an adversary $\mathcal{A}$ against the experiment $Exp_{\mathcal{A}}^{KeyExp}(\lambda)$, it could be used in the construction of another adversary $\mathcal{B}$ against the security experiment $Exp_{\mathcal{B}}^{SCDH}(\lambda)$. The adversary $\mathcal{B}$ works as follows: it receives as input an $X_4 \overset{\$}{\leftarrow} G$, and it needs to compute $SCDH_g(X_4)$. The adversary $\mathcal{B}$ proceeds as follows: It selects $x_1 \overset{\$}{\leftarrow} \mathbb{Z}_q$, $x_2, a \overset{\$}{\leftarrow} \mathbb{Z}_q^*$ and sets $X_1 = g^{x_1}$, $X_2 = g^{x_2}$, and $X_3 = X_4 * g^a$. That is, $\mathcal{B}$ implicitly sets $x_3 = \log_g X_3 = a + \log_g X_4$. Let $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ be a two-stage adversary. When

```
$Exp_{\mathcal{A}}^{KeyExp}(\lambda)$
$(G, \mathcal{D}, H) \leftarrow Setup(1^{\lambda})$
$g \xleftarrow{\$} G$
$X_4 \xleftarrow{\$} G$
$(\pi, state) \leftarrow \mathcal{A}_0(G, \mathcal{D}, H, g, X_4)$
$X_1, X_2, X_3 \xleftarrow{\$} G$
$\alpha \leftarrow CDH_g(X_1 * X_3 * X_4, X_2)^{\pi}$
$\beta \leftarrow CDH_g(X_1 * X_2 * X_3, X_4)^{\pi}$
$K_0 \leftarrow H(CDH_g(X_1 * X_3, X_2)^{\pi \cdot \log_g X_4})$
$K_1 \xleftarrow{\$} \{0,1\}^*$
$d \xleftarrow{\$} \{0,1\}$
$d' \leftarrow \mathcal{A}_{\infty}(state, X_1, X_2, X_3, \alpha, \beta, C_d)$
return $d \overset{?}{=} d'$
```

$\mathcal{A}_0$ is invoked with the corresponding parameters, it outputs the password π from the dictionary $\mathcal{D}$. $\mathcal{B}$ can compute $\alpha = (g^{x_1 x_2} * X_4^{2x_2} * g^{x_2 a})^{\pi}$. $\mathcal{B}$ samples random $\beta \xleftarrow{\$} G$. Now, $\mathcal{B}$ samples a string $\{0,1\}^*$, and assigns this to C_d. $\mathcal{A}_1$ is then invoked with all the necessary parameters. If $\mathcal{A}_0$ can identify d, it will have to query $CDH_g(X_1 * X_3, X_2)^{\pi \cdot \log_g X_4}$. $\mathcal{B}$ checks all the queries made by $\mathcal{A}_1$. From all the queries made by $\mathcal{A}_1$, $\mathcal{B}$ randomly picks one query Δ, and outputs $SCDH_g(X_4) = \frac{(\Delta^{1/x_2\pi})}{X_4^{(x_1+a)}}$. Now, we calculate the advantage of the adversary $\mathcal{B}$. Note that in this experiment we replace β with a random element from G. According to the DDH assumption, $CDH_g(X_1 * X_2 * X_3, X_4)$ is indistinguishable from random. Therefore, replacing it with a random element will reduce the advantage of $\mathcal{A}$ by a factor of $Adv_{\mathcal{A}}^{DDH}(\lambda)$. Let us assume that $\mathcal{A}_1$ makes Q queries to H. $\mathcal{B}$ selects one query made by $\mathcal{A}_1$ to H, and calculates $SCDH_g(X_4)$ on the basis of this. Thus, $Adv_{\mathcal{B}}^{SCDH}(\lambda) \geq 1/Q * (Adv_{\mathcal{A}}^{KeyExp}(\lambda) - Adv_{\mathcal{A}}^{DDH}(\lambda))$. That is, $Adv_{\mathcal{A}}^{KeyExp}(\lambda) \leq Adv_{\mathcal{A}}^{DDH}(\lambda) + Q * Adv_{\mathcal{B}}^{SCDH}(\lambda)$. Hence, the result holds.

4.4 Indistinguishability between the ideal and simulated worlds

The following theorem shows that the view of $\mathcal{Z}$ in the two worlds in Figure 5 is indistinguishable. For simplicity, we use $\mathcal{F}$ to refer to $\mathcal{F}_{APAKE}$.

Theorem 1. *Under the SCDH and DDH assumptions with access to the random oracle functionality $\mathcal{F}_{RO}$, the view of $\mathcal{Z}$ in the real world is indistinguishable from its view in the simulated world.*

Proof. Now, we show that the distinguishing advantage of $\mathcal{Z}$ between the real world and the simulated world is negligible. The argument uses a sequence of games, starting from the game in the real world and ending at the game in a simulated world. For any two adjacent games G_i and G_{i+1}, let $\mathbf{Dist}_{\mathcal{Z}}^{G_i,G_{i+1}}$ denote the distinguishing advantage of $\mathcal{Z}$ between them, i.e.,

$$\mathbf{Dist}_{\mathcal{Z}}^{G_i,G_{i+1}} = |\mathbf{Pr}[\mathcal{Z} \text{ outputs 1 in } G_i] - Pr[\mathcal{Z} \text{ outputs 1 in } G_{i+1}]|$$

Game G_0 and G_1: G_0 is the real world game. G_1 is almost the same as Game G_0, however, in this Game the simulator simulates all the NIZK proofs. Following Abdalla, Benhamouda and MacKenzie [1], we assume the adversary is algebraic: namely, the adversary is limited to performing only group operations on group elements in G (in Owl, the receiver is always required to check if a received public key is a proper group element). Under this assumption, Abdalla et al. have shown that Schnorr NIZK proofs are algebraic-simulation-sound extractable, and that the adversary's advantage is bounded by solving the discrete logarithm problem in the random oracle model. Hence, $\mathbf{Dist}_{\mathcal{Z}}^{G_0,G_1} = Adv_{\mathcal{A}}^{NIZK}(\lambda)$.

Game G_2: This game is almost similar to the Game G_1. However, in this game, we do not allow the protocol to use the same value of $x_1, x_2, x_4 \in \mathbb{Z}_q$. If two protocol instances come up with the same $g^{x_1}, g^{x_2}, g^{x_4}$, that was seen previously, the game halts and the adversary wins. The advantage of the adversary in Game G_2 will be different from the same in Game G_1 if we hit the birthday paradox. Thus, $\mathbf{Dist}_{\mathcal{Z}}^{G_1,G_2} = O\left(\frac{(T_{CLT}+T_{SVR})^2}{q}\right)$. Here, T_{CLT}, and T_{SVR} are the numbers of client-session and server-session queries respectively.

Game G_3: The difference between this game and Game G_2 is that in Game G_3, whenever the adversary $\mathcal{A}$ sends a `FlowTwo` message, then $\mathcal{SIM}$ first checks the correctness of the message. If the message is well-formed, $\mathcal{SIM}$ uses NIZK extraction to find the value of x_4 from Π_4 and the value of $b = x_4 \cdot \pi'$ from Π_α, and obtains $\pi' = b/x_4$. If there exists a record of the form $\langle FILE,\, P_i,\, P_j,\, \pi\rangle$, and $\pi = \pi'$, then return "correct guess". Mark the record $\langle ssid, P_j, P_i, \cdot\rangle$ as `compromised`. Else, send $(\texttt{TESTPWD}, sid, ssid, P_i, \pi')$ to $\mathcal{F}$, and pass its response to $\mathcal{A}$. If $\mathcal{F}$ replies with "correct guess", retrieve the record $\langle FILE, P_i, P_j, \cdot\rangle$, and replaces the last item with π'. Mark the record $\langle ssid, P_j, P_i, \cdot\rangle$ as `compromised`. If $\mathcal{F}$ returns "wrong guess", mark the record $\langle ssid, P_j, P_i, \cdot\rangle$ as `STALE`. It is easy to see that $\mathcal{Z}$'s view of G_2 and G_3 is the same, hence, $\mathbf{Dist}_{\mathcal{Z}}^{G_2,G_3} = 0$.

Game G_4: This game is the same as Game G_3 except the fact that in this game, when $\mathcal{A}$ sends a `FlowThree` message, $\mathcal{SIM}$ first checks the correctness of the message. If the message is correctly formatted and there was a previous `FlowOne` message, then $\mathcal{SIM}$ parses it to find $F = \langle \alpha, \Pi_\alpha\rangle$. $\mathcal{SIM}$ uses NIZK extraction to find the value of $a = x_2 \cdot \pi'$ from Π_α, and the value of x_2 from Π_2. $\mathcal{SIM}$ obtains $\pi' = a/x_2$. Then $\mathcal{SIM}$ checks if there is a record $\langle FILE, P_i, P_j, \pi\rangle$ marked `compromised`, where $\pi = \pi'$, then return "correct guess" to $\mathcal{A}$. Else, send $(\texttt{TESTPWD}, sid, ssid, P_i, \pi')$ to $\mathcal{F}$. If $\mathcal{F}$ returns "correct guess", then send "correct guess" to $\mathcal{A}$. Also mark the record $\langle ssid, P_i, P_j, \cdot\rangle$ as `compromised`. In all other cases, return "wrong guess" to $\mathcal{A}$, and mark the session record as `STALE`. Here, $\mathcal{Z}$'s view of G_3 and G_4 is the same. Thus, $\mathbf{Dist}_{\mathcal{Z}}^{G_3,G_4} = 0$.

Game G_5: This game is the same as Game G_4 except that when $\mathcal{SIM}$ has received a `FlowTwo` or a `FlowThree` message from $\mathcal{A}$, it checks if the corresponding session record is marked `compromised` or not. If $\mathcal{A}$ sends a `FlowTwo` or `FlowThree` message to $\mathcal{SIM}$, then if $\mathcal{SIM}$ had returned "correct guess" to $\mathcal{A}$, $\mathcal{SIM}$ selects a key k which is the same as what the adversary would have computed. If $\mathcal{SIM}$ had returned "wrong guess", then $\mathcal{SIM}$ selects k randomly from $\{0,1\}^\lambda$. From Lemma 3 we can say that if the adversary sends a `FlowThree` message to $\mathcal{SIM}$, then the difference between the advantages of the adversary in Game G_4 and Game G_5 is at most $Adv_{\mathcal{A}}^{RNDKey}(\lambda)$. That is to say that the difference will be at most $Q * Adv_{\mathcal{B}}^{SCDH}(\lambda) + \frac{1}{2} * Adv_{\mathcal{A}}^{DDH}(\lambda)$, where Q is the number of queries to $\mathcal{F}_{RO}$, $Q \in poly(\lambda)$. It is also evident that since the server uses a fixed element g^{x_3} and a variable x_4, if the simulator receives a `FlowTwo` message from $\mathcal{A}$, then the difference between the advantages of $\mathcal{Z}$ in Game G_4 and Game G_5 is at most $Adv_{\mathcal{A}}^{RNDKey1}(\lambda) \leq Q * Adv_{\mathcal{B}}^{SCDH}(\lambda) + \frac{1}{2} * Adv_{\mathcal{A}}^{MDDH}(\lambda)$, as in Lemma 4. Thus, $\mathbf{Dist}_{\mathcal{Z}}^{G_4,G_5} \leq poly(\lambda) * Adv_{\mathcal{B}}^{SCDH}(\lambda) + poly(\lambda) * Adv_{\mathcal{A}}^{DDH}(\lambda)$.

Game G_6**:** This game is the same as Game G_5 except that in this game when $\mathcal{SIM}$ receives a `FlowTwo` or a `FlowThree` message from an honest client or an honest server, it randomizes the session keys at both sides with both sides obtaining the same key. Lemma 5 proves that under such circumstances, the advantage gained by the adversary is negligible. That is to say $\mathbf{Dist}_{\mathcal{Z}}^{G_5,G_6}(\lambda) \leq Adv_{\mathcal{A}}^{DDH}(\lambda) + Q * Adv_{\mathcal{B}}^{SCDH}(\lambda)$.

Game G_7**:** This game is the same as Game G_6, except that in this game the protocol instance replaces the `FlowTwo` or `FlowThree` message with random elements from G. According to Assumption 2, the difference between the advantages of an adversary in Game G_6 and Game G_7 is at most $Adv_{\mathcal{A}}^{MDDH}(\lambda)$ when $\mathcal{SIM}$ sends a `FlowTwo` message and at most $Adv_{\mathcal{A}}^{DDH}(\lambda)$ when $\mathcal{SIM}$ sends a `FlowThree` message. In Lemma 1, we have shown that the DDH assumption is equivalent to the MDDH assumption. Therefore, $\mathbf{Dist}_{\mathcal{Z}}^{G_6,G_7} \leq O\left(Adv_{\mathcal{A}}^{MDDH}(\lambda)\right) = O\left(Adv_{\mathcal{A}}^{DDH}(\lambda)\right)$.

Game G_8**:** This game is the same as Game G_7 except that in Game G_8 when the adversary sends a $(\text{STEALPWFILE}, sid)$ query to $\mathcal{SIM}$, $\mathcal{SIM}$ sends this to $\mathcal{F}$. If $\mathcal{F}$ had returned "correct guess", and there must have been a query $(Hash, \langle sid, 1\rangle, \langle P_i, w\rangle)$ made to $\mathcal{F}_{RO}$ which was responded with t. Then send a query $(Hash, \langle sid, 2\rangle, t)$ to $\mathcal{F}_{RO}$ to obtain π, and return (π, g^t) to $\mathcal{A}$. If $\mathcal{F}$ had only returned "password file stolen", then it means there was no query $(Hash, \langle sid, 1\rangle, \langle P_i, w\rangle)$ made by $\mathcal{A}$, where w is the actual password. If $\mathcal{A}$ makes a query or causes a query of the form $(Hash, \langle sid, 1\rangle, \langle P_i, w\rangle)$, then make a OFFLINETESTPWD query to $\mathcal{F}$ for w. If $\mathcal{F}$ returns "correct password", $\mathcal{SIM}$ sends the query $(Hash, \langle sid, 1\rangle, \langle P_i, w\rangle)$ to $\mathcal{F}_{RO}$ and returns the response t to $\mathcal{A}$. Since this difference does not affect $\mathcal{Z}$'s view of the two worlds. As such, $\mathbf{Dist}_{\mathcal{Z}}^{G_7,G_8} = 0$. Game G_8 is the simulated world. This completes the proof.

5 Related work and comparison

In general, there are two ways to construct an augmented PAKE protocol: 1) by using a generic compiler to convert a balanced PAKE to an augmented one; 2) by designing an augmented PAKE specifically in the augmented setting.

Examples of the first approach include the A-method and the B-method. The A-method was first proposed by Bellovin and Merritt in 1993 [4]. It works by deriving a pair of digital signature keys from a password and saving only the public key at the server. The user holds only the password (and hence the private signing key derived from the password). This method was applied to convert EKE to an augmented version called A-EKE. The B-method was proposed by Jablon in 1997 to convert SPEKE to an augmented B-SPEKE scheme [24]. It works by adding a Diffie-Hellman protocol with the password being used as an ephemeral private key, in parallel to a PAKE protocol. Both A and B methods are generally applicable to convert a balanced PAKE to an augmented one. Similar methods were proposed by Gentry et al. based on digital signatures (called the π method) [12] and Haase et al. based on Diffie-Hellman [14]. Hwang et al. proposed two compilers based on the Diffie-Hellman scheme and a variant of the Schnorr signature in a round-efficient manner [21]. Although the generic methods have the advantage of working with any balanced PAKE, they *always* require more cost (rounds, computation or both), which makes them less appealing in practice. In Owl, we securely and efficiently combine Hwang et al's compiler [21] with a modified J-PAKE scheme, yielding an augmented PAKE protocol that provides extra security over J-PAKE against server compromise yet with lower computation overall.

The lack of efficiency in generic methods motivates specifically designing PAKE schemes in an augmented setting. Examples include KHAPE [13], OKAPE, aEKE [9], VTBSPEKE [32], KC-SPAKE2+ [35], PAK-Z+ [11], OPAQUE [26], AuCPace [14], SRP-6a [36], AMP [30] and AugPAKE [34]. Among these, KHAPE, OKAPE and aEKE all rely on an *ideal cipher* [3] (see Figure 5), an abstract building block that is assumed to not leak any information about the encryption content even when the encryption key has low entropy (e.g., using a password as the key). However, how to instantiate such an ideal cipher has remained an open problem [18]. VTBSPEKE and KC-SPAKE2+ rely on a *trusted setup* for all users but a compromise in this setup will break key exchange sessions for all users [32]. PAK-Z+ and OPAQUE rely on a hash-to-group (also called hash-to-curve in an EC setting) function, which is assumed to securely and efficiently map a password to a random generator of a designated group. However, instantiating this function is not straightforward as we will explain in more detail later. SRP-6a does not depend on an ideal cipher, a hash-to-group function or a trusted setup, which makes it relatively easier to implement than others. AMP and AugPAKE follow a similar design as SRP but both have questionable security. AMP has been repeatedly revised to patch vulnerabilities [31]. AugPAKE has not been published in a peer-reviewed paper (it is described in an IETF RFC), and its security proof is questioned by Jarecki et al. [26].

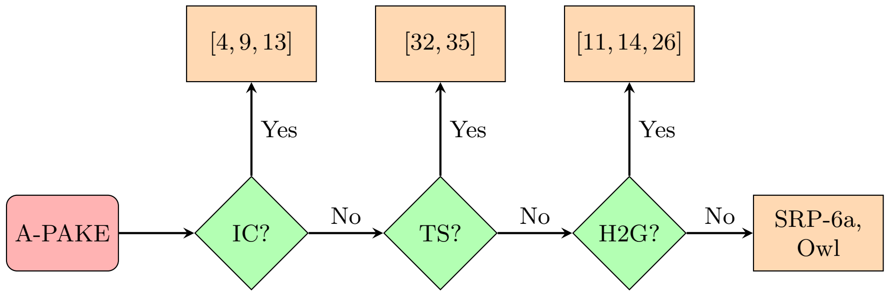

Fig. 6. Overview of selected augmented PAKE schemes and dependence on various assumptions. IC: Ideal Cipher. TS: Trusted Setup. H2G: Hash-to-Group

In this section, we focus on comparing Owl with SRP-6a and OPAQUE. We choose SRP-6a because it is the only augmented PAKE that has been widely used in practice. Although many provably secure augmented PAKE schemes have been proposed (see Figure 5), they generally rely on assumptions of an ideal cipher, a trusted setup or a hash-to-group function; however, difficulties in the realization of such assumptions have hindered the deployment of these schemes [18]. We choose OPAQUE because it was chosen by the IETF as a candidate for standardization and a possible replacement for SRP-6a.

Hash-to-group/hash-to-curve. In the original OPAQUE paper [26], the authors specify the protocol only in an EC setting, not in any MODP group. The rationale was however not explained. To provide insights into this design choice, we first need to clarify the hash-to-group (H2G) and hash-to-curve (H2C) functions. OPAQUE critically relies on H2C to map a password to a random prime-order generator on an elliptic curve in *constant time*. H2G is the equivalent function in a MODP group. The original idea of using H2G in PAKE is from Jablon's 1996 design of SPEKE [23]. More concretely, SPEKE defines a safe prime $p = 2q + 1$ as the modulus, i.e., q is also prime. The H2G function with input a is defined as $f(a) = a^2 \bmod p$, which forms the basis of SPEKE as standardized in IEEE 1363.2 [22]. However, using a safe prime as modulus makes modular exponentiations costly. Extending this construction to a DSA group (which uses short exponents) and EC proves harder than it seems.

- **DSA**. For a DSA group (p, q, g), IEEE 1363.2 defines the H2G function with input a as: $f(a) = a^{(p-1)/q}$ [22]. This function has two known issues: 1) it is costly due to the long exponent; 2) it may return an identity element (called an 'invalid' output). IEEE 1363.2 does not define any handling for 'invalid' outputs; handling such exceptions can forfeit the constant-time property.
- **EC**. To do the equivalent of H2G in EC, IEEE 1363.2 defines an H2C function based on a trial-and-increment method. The defined H2C function guarantees the output is a prime-order generator (no 'invalid' output) and works with general curves, but is vulnerable to timing attacks [37]. IEEE 1363.2 was officially withdrawn in 2019. Since 2018, IETF has been trying to define

custom-built H2C functions for selected curves in an internet draft [20], but existing functions do not exclude low-order (i.e., 'invalid') points by design and may not work with future curves [18]. Whether these functions operate in constant time remains to be established (they must be fully defined first).

To date, the fully defined and generally accepted mapping function in MODP is the one specified in the original SPEKE paper [23]. As we will explain, applying the same function to OPAQUE would make it far less efficient than SRP-6a. All these issues in MODP are avoided if we only consider EC. The OPAQUE authors assume a secure and efficient H2C function is available for elliptic curves but without concretely instantiating it in the paper. The lack of a complete specification of OPAQUE makes it difficult to directly compare Owl with OPAQUE; we have to treat H2C as an abstract block as well as filling in details of H2G in a MODP setting for a comprehensive comparison. On the other hand, SRP-6a is fully defined in MODP with a safe-prime modulus, which allows us to directly compare it with Owl. Details of the comparison are presented below.

Computational performance. SRP-6a mandates the use of a safe prime $p = 2q + 1$ where q is also a prime as the modulus. This leaves the protocol undefined for a multiplicative group with short exponents such as DSA, as well as for an elliptic curve setting. OPAQUE is built on two building blocks: 1) authenticated key exchange (AKE) and 2) hash-to-curve. After OPAQUE was selected by IETF, its designers modified the protocol by using 3DH instead of HMQV to instantiate AKE in an IETF Internet draft [28]. Here, we will evaluate the performance of OPAQUE based on using HMQV as specified in the original paper [26]. (The performance will decrease when 3DH is used). The OPAQUE paper assumes a hash-to-curve function to map a password to a generator of the prime-order (sub)group on an elliptic curve, which leaves the protocol undefined for the MODP setting. We fill this gap by using two known hash-to-group functions as defined for SPEKE and PAK in IEEE 1363.2 [22] to do the equivalent of hash-to-curve in two MODP settings respectively: 1) using a safe-prime modulus; 2) using DSA groups. Table 1 summarizes the comparison results. Owl has clear advantages over SPR-6a in terms of flows, computation, and agility to work with both MODP and EC settings. Implementing OPAQUE in DSA gives a lower computation cost than using a safe-prime modulus (at the expense of having the 'invalid' output issue). In the DSA setting, the client requires more computation than Owl, but the server requires less. It is however difficult to directly compare these two in the EC setting due to H2C not having been concretely instantiated, hence the cost of H2C is yet unknown (aside from the 'invalid' output and the agility issues that need to be addressed).

Round efficiency. Owl only requires one flow in the registration process, fewer than others. Between Owl and OPAQUE, OPAQUE seems to have the advantage of needing only two flows in the login process; Owl needs three. However, with only two flows, it is impossible for OPAQUE to complete client-to-server authentication in a login process, as later pointed out by Bradley, Jarecki and Xu [5]. The (theoretical) advantage of a two-flow OPAQUE is that it allows the server to be authenticated in the second flow based on explicit key confirmation

Scheme	Flows	KC	MODP (safe prime)		MODP (DSA)		Elliptic Curve	
			Client	Server	Client	Server	Client	Server
SRP-6a	3/4	Exp	2(x12)+1	2(x12)+1	–	–	–	–
OPAQUE	3/2	Imp	4.5(x12)	3.5(x12)	1(x11)+6.5†	5.5†	4.5+H2C‡	3.5
Owl	1/3	Imp	–	–	14	13	11	10

Table 1. Comparison among selected PAKE schemes. The flows column gives the number of flows for registration/login respectively. Computational cost columns give the number of exponentiation in the MODP setting, assuming a 3072-bit modulus for concreteness. 2(×12) denotes that the cost of one exponentiation (3071-bit exponent) is about the same as 12 typical 3072-bit DSA group exponentiations (256-bit exponent). The hash-to-group function in OPAQUE requires an exponentiation with a 2816-bit exponent (co-factor), which has cost equal to 11 exponentiations with 256-bit exponent in DSA. H2C denotes (yet unknown) cost of a hash-to-curve function. †denotes having concerns about 'invalid' output. ‡denotes having concerns not only about the cost, but also 'invalid' output and agility.

(before the client is authenticated). However, doing so will expose the server to an *undetectable* online dictionary attack [27], in which an attacker (client) does not send the third flow and the server cannot distinguish the authentication failure from a drop-out. (In an asymmetric setting, the server is always online responding to requests, and hence is more vulnerable to online dictionary attacks.) Preventing this attack requires the client to be authenticated first; in this case, OPAQUE and Owl have the same round efficiency.

Message size. We now compare the sizes of the messages among the three protocols. For simplicity, we only consider the login phase and focus on public-key cryptographic data (excluding auxiliary data such as user identities). Furthermore, we only consider implicit key confirmation, hence excluding explicit key confirmation strings which apply to all schemes. Table 2 summarizes the result. SRP-6a [36] involves exchanging two group elements in the login process (we omit the salt). Assume a 3072-bit safe-prime modulus is used. The total size is $3072 \times 2 = 6{,}144$ bits $= 768$ bytes. OPAQUE [26] involves exchanging four group elements and an encrypted string using an authenticated encryption scheme (containing two group elements and a private key generated from the registration phase). Assume OPAQUE uses the same 3072-bit safe-prime modulus as SRP-6a. The total size is approximately $3072 \times 6 + 3071 = 21{,}503$ bits $=$ 2,688 bytes (we omit the size of the IV and the message authenticated code). If a 3072-bit DSA group is used instead, the total size is $3072 \times 6 + 256 = 18{,}688$ bits $= 2{,}336$ bytes. Assume an EC setting where the group has a prime order of 256 bits. We assume a group element requires 257 bits in a compressed form. Hence, the size of the messages is $257 \times 6 + 256 = 1{,}798$ bits $= 225$ bytes. Owl involves sending 6 group elements, 6 Schnorr ZKP and a short response in the key exchange process. In a 3072-bit DSA group, the size of each ZKP (h, r) is 512 bits (see §3.1). Hence, the total size is $3072 \times 6 + 512 \times 6 + 256 = 21{,}760$ bits $= 2{,}720$ bytes. In an EC setting, the size is $257 \times 6 + 512 \times 6 + 256 = 4{,}870$ bits

Schemes	MODP (safe prime)	MODP (DSA)	Ellipic Curve
SRP-6a	768 B	–	–
OPAQUE	2688 B	2336 B	225 B
Owl	–	2720 B	609 B

Table 2. Comparison of message sizes in bytes (B) among selected PAKE schemes. The 'MODP (safe prime)' column gives the size in a MODP setting, assuming a 3072-bit safe-prime modulus. The 'MODP (DSA)' column gives the size in a MODP setting, assuming a 3072-bit DSA group with a 256-bit exponent. The "Elliptic Curve" column gives the size in an EC setting, assuming a 256-bit prime-order EC group.

= 609 bytes. Although Owl involves sending more group elements than SRP-6a, it can actually use less bandwidth because of its agile support for EC implementations. In the same EC setting, the size of the messages in Owl is bigger than that in OPAQUE, but Owl is more flexible to work with any elliptic curve that is suitable for cryptography while OPAQUE is restricted by the availability of the hash-to-curve function on that curve.

Password update. In every login session in OPAQUE, the server sends a pre-computed ciphertext using authenticated encryption and a password-derived encryption key. When the password is changed, the pre-computed ciphertext will change, hence revealing whether the password has been changed from the last login. By comparison, SRP-6a and Owl do not have this issue.

Server compromise. When the password verification files stored on a server are stolen, all three schemes will be vulnerable to an offline dictionary attack. OPAQUE has an advantage in that the attacker cannot use any pre-computed table. By contrast, for SRP-6a and Owl, it is possible to use a pre-computed table, but the attacker must generate a *unique* pre-computed table for each user. This requires significant computation and storage, which makes the pre-computation attack less practical. It is possible to add *pre-computation security* by using an OPRF-based compiler as proposed by Jarecki et al. [26], but it introduces the reliance on H2C and the related implementation issues in both MODP and EC settings. Motivated by removing the reliance on H2C, Bradley, Jarecki and Xu [5] recently proposed an alternative method to achieve pre-computation security, but their method depends on a *trusted setup*, which introduces trapdoor concerns. How to achieve pre-computation security without incurring such issues (e.g., reliance on H2G, TS, or IC; see Figure 5) is an open question which we leave for future research.

6 Conclusion

In this paper, we present a new augmented PAKE protocol called Owl. Our design strategy is to utilize Schnorr non-interactive zero-knowledge proof to enforce every party to follow the protocol specification honestly. We formally prove the security of Owl based on the standard DDH and CDH assumptions under the UC framework in the random oracle model. Through a thorough evaluation, we show Owl is the first protocol that provides systematic advantages over SRP-6a in terms of provable security, computation, bandwidth and agility for implementation in versatile group settings. Owl's agility is attributed to the fact that it utilizes only the most basic addition/multiplication operations on an elliptic curve, or equivalently, multiplication/exponentiation in MODP, plus a *standard* one-way hash function for implementation without depending on any ideal cipher, trusted setup, hash-to-group functions or special group characteristics (such as the mandated use of a safe-prime modulus in SRP-6a).

OPAQUE was chosen by the IETF for reasons of both efficiency and SRP-6a's lack of security proofs. However, implementing a MODP version of OPAQUE runs into issues in terms of instantiating a hash-to-group function, while implementing OPAQUE in EC depends on a not yet fully instantiated hash-to-curve function. Broad adoption of OPAQUE may also depend on a remedy for the issue of signalling password changes to passive observers (§2), which may require protocol modifications (and revision of security proofs). While these open issues in OPAQUE are being addressed, we believe the availability of alternatives such as Owl, with clear advantages over SRP-6a, are useful to the community.

References

1. M. Abdalla, F. Benhamouda, and P. MacKenzie. Security of the J-PAKE password-authenticated key exchange protocol. In *IEEE S&P*, 2015.
2. F. Bao, R. Deng, and H. Zhu. Variations of diffie-hellman problem. In *ICICS*, 2003.
3. M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In *Eurocrypt*, 2000.
4. S. Bellovin and M. Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In *CCS*, 1993.
5. T. Bradley, S. Jarecki, and J. Xu. Strong asymmetric pake based on trapdoor ckem. In *Crypto*. Springer, 2019.
6. R. Canetti, S. Halevi, J. Katz, Y. Lindell, and P. MacKenzie. Universally composable password-based key exchange. In *Eurocrypt*. Springer, 2005.
7. J. Coron, A. Gouget, T. Icart, and P. Paillier. Supplemental access control (PACE v2): Security analysis of PACE integrated mapping. In *Cryptography and Security: From Theory to Applications*, pages 207–232, 2012.
8. Crypto Forum Research Group (CFRG). IETF PAKE selection. `https://github.com/cfrg/pake-selection`, 2020.
9. B.F. Dos Santos, Y. Gu, S. Jarecki, and H. Krawczyk. Asymmetric pake with low computation and communication. In *Eurocrypt*. Springer, 2022.
10. A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Crypto*, 1987.
11. C. Gentry et al. PAK-Z+. *Submission to IEEE P1363.2*, 2005.
12. C. Gentry, P. MacKenzie, and Z. Ramzan. A method for making password-based key exchange resilient to server compromise. In *Crypto*, 2006.
13. Y. Gu, S. Jarecki, and H. Krawczyk. KHAPE: Asymmetric PAKE from key-hiding key exchange. In *Crypto*, 2021.

14. B. Haase and B. Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR Trans. CHES*, pages 1–48, 2019.
15. F. Hao. Schnorr non-interactive zero-knowledge proof. *RFC 8235*, 2017.
16. F. Hao. Prudent practices in security standardization. *IEEE Communications Standards Magazine*, 2021.
17. F. Hao and P. Ryan. Password authenticated key exchange by juggling. In *SPW*, 2008.
18. F. Hao and P.C. van Oorschot. SoK: Password-Authenticated Key Exchange–Theory, Practice, Standardization and Real-World Lessons. In *AsiaCCS*, 2022.
19. D. Harkins. Dragonfly Key Exchange. *RFC 7664*, 2015.
20. A. Hernández, S. Scott, N. Sullivan, R. Wahby, and C. Wood. Hashing to elliptic curves. *IETF draft-irtf-cfrg-hash-to-curve-16*, 2023.
21. J Hwang, S. Jarecki, T. Kwon, J. Lee, J. Shin, and J. Xu. Round-reduced modular construction of asymmetric password-authenticated key exchange. In *SCN*. Springer, 2018.
22. IEEE Standards Association. IEEE 1363.2-2008: IEEE Standard Specification for Password-Based Public-Key Cryptographic Techniques. `https://standards.ieee.org/standard/1363_2-2008.html`.
23. D. Jablon. Strong password-only authenticated key exchange. *SIGCOMM Computer Commun. Review*, 26(5):5–26, 1996.
24. D. Jablon. Extended password key exchange protocols immune to dictionary attack. In *IEEE Workshop on ETICE*, 1997.
25. D. Jablon. Password authentication using multiple servers. In *CT-RSA*, 2001.
26. S. Jarecki, H. Krawczyk, and J. Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In *Eurocrypt*, 2018.
27. Y. Kobayashi et al. Gateway threshold password-based authenticated key exchange secure against undetectable on-line dictionary attack. In *ICETE*. IEEE, 2015.
28. H. Krawczyk, D. Bourdrez, K. Lewi, and C. Wood. The OPAQUE asymmetric PAKE protocol. *IETF draft-irtf-cfrg-opaque-9*, 2022.
29. K. Kurosawa and R. Nojima. Simple adaptive oblivious transfer without random oracle. In *AsiaCrypt*, pages 334–346. Springer, 2009.
30. T. Kwon. Authentication and key agreement via memorable password. In *NDSS*, 2001.
31. T. Kwon. Revision of AMP in IEEE P1363.2 and ISO/IEC 11770-4. *Submission to IEEE P1363*, 2005.
32. D. Pointcheval and G. Wang. VTBPEKE: Verifier-based two-basis password exponential key exchange. In *AsiaCCS*, 2017.
33. S. Shin and K. Kobara. Efficient augmented password-only authentication and key exchange for IKEv2. *RFC 6628*, 2012.
34. S. Shin, K. Kobara, and H. Imai. Security proof of AugPAKE. *IACR ePrint*, 334, 2010. See also [33].
35. V. Shoup. Security Analysis of SPAKE2+. In *TCC*, 2020.
36. SRP Protocol Design. `http://srp.stanford.edu/design.html`. Includes description of SRP-6a. Accessed 10 February 2023.
37. M. Vanhoef and E. Ronen. Dragonblood: Analyzing the Dragonfly handshake of WPA3 and EAP-pwd. In *IEEE S&P*, 2020.
38. T. Wu. The Secure Remote Password protocol. In *NDSS*, 1998.