Making an Asymmetric PAKE Quantum-Annoying by Hiding Group Elements[‡]

Marcel Tiepelt[1] [0000−0002−3389−208X], Edward Eaton[2], and Douglas Stebila[3] [0000−0001−9443−3170]

[1] KASTEL, Karlsruhe, Germany, `marcel.tiepelt@kit.edu`
[2] National Research Council Canada, Ottawa, Ontario, Canada
[3] University of Waterloo, Waterloo, Ontario, Canada, `dstebila@uwaterloo.ca`

Abstract The KHAPE-HMQV protocol is a state-of-the-art highly efficient asymmetric password-authenticated key exchange protocol that provides several desirable security properties, but has the drawback of being vulnerable to quantum adversaries due to its reliance on discrete logarithm-based building blocks: solving a single discrete logarithm allows the attacker to perform an offline dictionary attack and recover the password. We show how to modify KHAPE-HMQV to make the protocol *quantum-annoying*: a classical adversary who has the additional ability to solve discrete logarithms can only break the protocol by solving a discrete logarithm for each guess of the password. While not fully resistant to attacks by quantum computers, a quantum-annoying protocol could offer some resistance to quantum adversaries for whom discrete logarithms are relatively expensive. Our modification to the protocol is small: encryption (using an ideal cipher) is added to one message. Our analysis uses the same ideal cipher model assumption as the original analysis of KHAPE, and quantum annoyingness is modelled using an extension of the generic group model which gives a classical adversary a discrete logarithm oracle.

Keywords: password-authenticated key exchange · quantum-resistant · quantum-annoying · generic group model

1 Introduction

A wide-spread method for authentication in client-server situations involves a key exchange where the server is authenticated through a public key infrastructure,

[‡]This version of the contribution has been accepted for publication, after peer review but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record of this contribution is published in ESORICS 2023, Lecture Notes in Computer Science, vol 14344, available online at: https://doi.org/10.1007/978-3-031-50594-2_9. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

while the client authenticates themselves with a password by transmitting the password directly over the encrypted channel. This method is suboptimal since the user's password is exposed to the server.

A *password authenticated key exchange* (PAKE) protocol enables two parties to perform a key exchange, authenticated using mutual knowledge of a shared password, without revealing the password to the network or to each other. The setting of PAKEs allows two kinds of attacks: online attacks (the adversary interacting with either party), and offline attacks (the adversary operating locally based on what is has observed from previous online interactions). Password-based protocols are always vulnerable to online dictionary attacks, where the adversary can rule out one password guess with each online interaction with a party. The goal of a PAKE is to ensure that offline dictionary attacks are infeasible, for example because of an intractability assumption. While PAKEs have been known for decades, there was little progress in adoption for many years, but there is renewed interest in adoption of PAKEs via a variety of recent and ongoing standardization efforts [18,6,19,1,2].

This paper focuses on KHAPE [10], a compiler that turns a *key hiding* authenticated key exchange (KH-AKE) and a PAKE into an *asymmetric* PAKE (aPAKE). Asymmetric PAKEs improve upon regular PAKEs by forcing an attacker to perform an exhaustive search on the password even after server compromise, since the value stored by the server cannot be used to impersonate the client. The OPAQUE framework [13] introduced the notion of *strong* asymmetric PAKEs (saPAKE), which further guarantees that no pre-computation can be performed to aid in the exhaustive search for the password in the case of server compromise. This is achieved by combining an oblivious pseudo-random function (OPRF) and a PAKE.

Most PAKEs are based on the hardness of solving the discrete logarithm problem (see [11] for an overview), making them vulnerable to attacks by quantum computers, thus motivating the question of building PAKEs that are quantum-resistant. The obvious answer is to build new PAKEs that rely on post-quantum intractability assumptions, and post-quantum PAKEs are starting to emerge in the literature. These new PAKEs (*e.g.*, see [4]) are based on key encapsulation mechanisms to match the standardized quantum-secure encryption [15]. However, there may be other interim options requiring fewer modifications by augmenting existing protocols.

Quantum-annoying PAKEs. During the CFRG PAKE standardization process in 2019, it was observed [20] for one of the Diffie–Hellman-based candidates that even if an attacker could solve discrete logarithms, they could not immediately recover the password. Instead, an attacker seemed to have to do a discrete logarithm for each guess of the password even during an offline dictionary attack: this property was named "quantum-annoying". If solving such a problem remains reasonably expensive, then a moderate level of security can still be achieved.

Eaton and Stebila [7] developed a formalization of the quantum-annoying property for PAKEs by considering a classical adversary working in the generic group model who is given the additional power of a discrete logarithm oracle.

They showed that the base version of the symmetric PAKE protocol CPace [3] was quantum-annoying in the generic group model. One main characteristic of CPace that lead to it being quantum-annoying is that the password π shared by the client and server is used to derive a generator g_π of the group, and then a Diffie–Hellman key exchange is performed using that generator (g_π^{xy}). But from the perspective of an adversary who only sees Diffie–Hellman public keys (g_π^x and g_π^y), no information is gained about the password π since for each π' there is an x' such that $g_{\pi'}^{x'} = g_\pi^x$.

Our contributions. Whereas CPace is a symmetric PAKE, the KHAPE-HMQV protocol constructed by the KHAPE compiler [10] is an asymmetric PAKE, so compromise of a server using KHAPE-HMQV does not enable the adversary to impersonate a user without first performing an offline dictionary attack. However, the protocol is not quantum-annoying: after seeing just a single transcript, a single discrete logarithm computation suffices to enable an offline dictionary attack to recover the user's password. We address this vulnerability by presenting the QA-KHAPE protocol, a quantum-annoying variant of KHAPE-HMQV. As shown in Fig. 1, our modifications entail encapsulating an additional key into the server-stored credentials, which is later used by the principals to encrypt their Diffie–Hellman key-pairs prior to exchanging messages. This effectively means that each guess of the password causes the transcript to decrypt (under a symmetric key dependent on the password) to a different pair of Diffie–Hellman public keys, so a new discrete logarithm must be performed each time.

The changes to the protocol require only minimal computational and communication overhead, with the same number of rounds as KHAPE-HMQV and only a single additional ideal cipher ciphertext (increasing the server-client ciphertext from three to four elements). The client-server communication remains unchanged, and the protocol requires two additional ideal cipher computations, one encryption, and one decryption.

We show that QA-KHAPE is quantum-annoying following the methodology of [7]: the adversary is a classical adversary in the generic group model with the addition of a discrete logarithm oracle. In Section 3.1, we define a security game in the generic group model tailored to capturing the core quantum annoying property of the QA-KHAPE protocol. In Section 4, we apply this to show that QA-KHAPE is secure in a quantum-annoying variant of the standard Bellare–Pointcheval–Rogaway (BPR) security model for asymmetric password authenticated key exchange.

Limitations. Just as in the original security proof of KHAPE by [10], our analysis also relies on the ideal cipher assumption. Care must be taken for an instantiation of the IC, which is discussed in [10, Section 8].

Further, we wish to highlight for the reader that the "quantum annoying" security notion is an intermediate notion below fully quantum-resistant. One limitation of the quantum annoying security notion is that it has a narrow view of quantum capabilities: by using a formalism in the generic group model with a discrete logarithm oracle, we are effectively assuming that the only quantum operation an adversary will do is run Shor's algorithm, which is certainly less than the full power available to a polynomial time quantum computer.

Even just considering security against quantum computers running Shor's algorithm, a protocol "secure" in the quantum-annoying model is still vulnerable to attacks by quantum computers, it is just that the attack scales in the size of the password space. This leads to the question of the cost of computing a discrete logarithm on a quantum computer. While it is impossible to predict the efficiency of quantum computers in the far future, current research suggests that the first generations of quantum computers capable of solving cryptographically relevant discrete logarithm problems will require significant resources in order to do so [17,8,9,16]. These estimates are undoubtedly coarse and may be off by several orders of magnitude, but it is plausible that even for early cryptographically relevant quantum computers, computing a single discrete logarithm will not be cheap, and that computing millions of discrete logarithms to find the password in a quantum-annoying PAKE may be prohibitively expensive.

More recently, a preprint has examined the "multiple discrete logarithm" problem induced by the quantum annoying model [12]. In this work, the authors show that it is possible to (asymptotically) solve m discrete logarithm problems (in a generic group) with a quantum computer more efficiently than m times the cost of a single Shor's instance. In particular, their algorithm solves m discrete logarithms with around $\log m$ times fewer quantum group operations (if $m = \Omega(\log p)$, where p is the size of the group). This comes at the expense of requiring large quantum memory to compute everything simultaneously. Whether this represents a concrete improvement to the ability of an adversary to break quantum annoying security (and if so, how large the grouping m should be) is an interesting open question. Our proofs bound the adversary's success probability in terms of the number of discrete logarithm oracle queries made. If it is a practical improvement to group such queries, this does not affect our proofs, only how the induced bounds translate to real-world estimates of adversary cost.

2 Preliminaries

2.1 Quantum Annoying-ness in the Generic Group Model.

In the normal generic group model there is a multiplicative public representation of group elements taken uniformly from $\{0,1\}^\kappa$, and an additive secret representation in $\mathbb{Z}_p$. The public representations have no intrinsic structure, and so any information about the group is obtained through the group operation oracle. Let $\langle g \rangle = \mathbb{G}$ be a generic group of size p with group operation $\circ$. When $g_w \circ g_v$ is queried, for example $(g^v, g^w) \mapsto g^{v+w}$, a table $\mathcal{T}_{\mathrm{ggm}}$ is used to retrieve the secret representations of g_v and g_w, $v, w \in \mathbb{Z}_p$. Then $v + w \pmod p$ is the secret representation of g^{v+w}. If g^{v+w} has already been given a public representation, that is returned. Otherwise, a uniformly random string is sampled from $\{0,1\}^\kappa$, assigned as a new public representation to g^{v+w} in the table $\mathcal{T}_{\mathrm{ggm}}$, and provided back to the querier. Similarly, the discrete logarithm oracle $\textsc{Dlog} : \mathbb{G} \times \mathbb{G} \to \mathbb{Z}_p$ takes as input two group elements and outputs the discrete logarithm. The query

$\textsc{Dlog}(g_v, g_w)$ can be responded to by looking up g_v and g_w in $\mathcal{T}_{\text{ggm}}$ and returning $w \cdot v^{-1} \pmod p$.

The generic group model is a powerful tool, but limited in its ability to reason about whether the adversary's interactions with the discrete logarithm oracle are sufficient to determine $\textsc{Dlog}(g, g_t)$ for a specific group element g_t. Naturally, if they have made exactly this query, the discrete logarithm is known. But other queries, such as $\textsc{Dlog}(g, g_t^2)$, are also sufficient to make $\textsc{Dlog}(g, g_t)$ knowable.

The framework of [7] simulates the generic group model in such a way that such specific statements can be made. Let $G_1, G_2, \dots G_\mu$ be a collection of (public representations of) group elements whose discrete logarithm (with respect to the group generator g) are of potential interest to the adversary. When we maintain the group, rather than imbuing these group elements with specific secret representations in $\mathbb{Z}_p$, we instead denote each as a formal independent variable $\chi_1, \dots, \chi_\mu$. Group operations now correspond to addition over a *vector space* of dimension $\mu+1$. For example, in computing $(G_1 \circ G_1) \circ G_2$ we would calculate the secret representation as $2\chi_1 + \chi_2$ and give this a unique public representation in $\{0,1\}^\kappa$. Thus, secret representations can now be written as a linear combination of the χ_i variables, *i.e.*, $\alpha_0 + \sum_i \alpha_i \chi_i$.

Thinking about how these secret representations interact with the $\textsc{Dlog}$ oracle is how we can start to reason about what discrete logarithms are. Say the adversary queries $\textsc{Dlog}(A, B)$, and the secret representation of A is $\alpha_0 + \sum \alpha_i \chi_i$ (respectively with β for B). If the adversary is given the response δ (so that $A^\delta = B$), this imposes a constraint on our variables. Specifically, it says that $\delta\left(\alpha_0 + \sum \alpha_i \chi_i\right) = \beta_0 + \sum \beta_i \chi_i$, which we can rewrite as

$$\sum_{i=1}^{\mu} (\delta\alpha_i - \beta_i)\chi_i = \beta_0 - \delta\alpha_0. \tag{1}$$

This linear constraint lets us define an equivalence relation: if two secret representations are the same 'modulo' the linear constraints imposed by responses to $\textsc{Dlog}$, they should have the same public representation. Consequently, if, modulo these constraints, a secret representation χ_i is equivalent to some $a \in \mathbb{Z}_p$, then $\textsc{Dlog}(g, G_i)$ has taken on a definite value a, whether or not it was actually queried. Otherwise, it can still take on any possible value.

By taking the coefficients of the χ_i variables in Equation 1 we can construct a matrix D and a vector $\vec{r}$ (we write vectors as column vectors), so that the set of constraints is easily summarized as $D\vec{\chi} = \vec{r}$. Similarly, a secret representation $a_0 + \sum a_i \chi_i$ can be written as the pair $(a_0, \vec{a})$. In more detail, the equivalence relation can be defined as follows:

Definition 1. *For group elements g_a, g_b with secret representation $(a_0, \vec{a})$ and $(b_0, \vec{b})$, we say that g_a is* $(D, \vec{r})$-equivalent *to g_b if there exists an $\vec{\omega} \in \mathbb{Z}_p^{q_D}$ such that $\vec{\omega}^T D = \vec{a}^T - \vec{b}^T$ and $\vec{\omega}^T \vec{r} = b_0 - a_0$.*

Note that this is indeed an equivalence relation (reflexivity is proven by taking $\vec{\omega} = \vec{0}$, symmetry is proven by taking $-\vec{\omega}$, and transitivity is proven by taking

$\vec{\omega}_1 + \vec{\omega}_2$). The reason that this definition gives us what we want is that when it is satisfied, we have that $b_0 - a_0 = \vec{\omega}^T \vec{r} = \vec{\omega}^T D \vec{\chi} = (\vec{a}^T - \vec{b}^T)\vec{\chi} = \vec{a}^T \vec{\chi} - \vec{b}^T \vec{\chi}$, telling us that $a_0 + \sum a_i \chi_i = b_0 + \sum b_i \chi_i$, as we expect. We can now describe how the $\mathbb{G}$ and the DLOG oracle are simulated in full detail. Note that the simulation is not *efficient* [7, Sec. 4], since the simulation requires to search through all previous queries to check if a linear relationship exists. However, the purpose of the framework is to give an information-theoretic bound (in the generic group model) relative to the number of discrete logarithm queries to define a specific discrete logarithm, thus the exact efficiency is not relevant.

DLOG(g_V, g_W): If g_V or g_W do not exist in T_{ggm}, then abort. Otherwise, let $(v_0, \vec{v}), (w_0, \vec{w})$ be secret representations of g_V, g_W respectively. Sample a random vector $\vec{s}$ such that $D\vec{s} = \vec{r}$ and compute $\delta = (w_0 + \langle \vec{w}, \vec{s} \rangle)/(v_0 + \langle \vec{v}, \vec{s} \rangle) \mod p$. Add the row $\delta \vec{v}^T - \vec{w}^T$ to D, and value $w_0 - \delta v_0$ to vector $\vec{r}$. Then δ is the discrete logarithm that is returned. This corresponds to [7, Alg. 2].

$\circ(g_V, g_W)$: If g_V or g_W do not exist in T_{ggm}, then abort. Otherwise, for the secret representations $(v_0, \vec{v})$, $(w_0, \vec{w})$, let $(z_0, \vec{z}) = (v_0 + w_0, \vec{v} + \vec{w})$. If z appears in T_{ggm}, return the corresponding public representation. Otherwise, check if there exists an entry $(f_0, \vec{f})$ of T_{ggm} that is $(D, \vec{r})$-equivalent to $(z_0, \vec{z})$. If so, return the public representation of that entry. If no such $(D, \vec{r})$-equivalent entry exists, sample a new public representation, add the entry $T_{\text{ggm}}[g_Z] = (z_0, \vec{z})$ and return g_Z. This corresponds to [7, Alg. 5].

With this setup, we can prove Lemma 1, which is a generalization of [7, Lemma 1] and an instantiation of which is used in a game hop in Section 3.2.

Lemma 1 (Unique Solutions). *Let g_a and g_b be public representations of group elements, with corresponding secret representations $(a_0, \vec{a}), (b_0, \vec{b})$. Let $(D, \vec{r})$ be the current set of constraints on discrete logarithms. Then the discrete logarithm of g_b with respect to g_a is defined if and only if $[\vec{b}^T | b_0]$ is in the rowspace of the matrix* $\left[\begin{array}{c|c} -D & \vec{r} \\ \hline \vec{a}^T & a_0 \end{array}\right]$.

Proof. The discrete logarithm is defined if and only if there exists an α such that g_a^α is $(D, \vec{r})$-equivalent to g_b. By definition, this is the same as the existence of $\alpha, \vec{\omega}$ such that $\vec{\omega}^T D = \alpha \vec{a}^T - \vec{b}^T$, and $\vec{\omega}^T \vec{r} = b_0 - \alpha a_0$. We can rewrite this relation as $\left[\vec{b}^T \mid b_0\right] = \left[-\vec{\omega}^T D + \alpha \vec{a}^T \mid \vec{\omega}^T \vec{r} + \alpha a_0\right] = \left[\begin{smallmatrix} \vec{\omega} \\ \alpha \end{smallmatrix}\right]^T \left[\begin{array}{c|c} -D & \vec{r} \\ \hline \vec{a}^T & a_0 \end{array}\right]$.

This establishes that if the discrete logarithm is defined, $[\vec{b} \mid b_0]$ is indeed in the rowspace, and if it is in the rowspace that the discrete logarithm is defined (and equal to the α value that is the scalar for the 'a' row). □

Corollary 1. *Let g_b be the public representation of a group element and $(b_0, \vec{b})$ the corresponding secret representation. Let g be the generator of the group, which has secret representation $(1, \vec{0})$. Then the discrete logarithm of g_b with respect to g is defined if and only if $\vec{b}$ is in the row span of D.*

Proof. We apply Lemma 1 with $\vec{a} = \vec{0}$. Since the zero vector cannot affect the row span, we can conclude that $\vec{b}^T$ must be in the row span of D.

Table 1. Examples for simulation of queries to the generic group model group operation and the ideal cipher. The queries are in order from top to bottom. The public representations returned from the oracle are uniformly random strings that contain no information beyond what was given in the query and particularly are sampled independently.

Oracle	Label	Public Representation	Secret Representation
Initialization	g_0	11001100110111101	1
$\circ(g_0, g_0)$	$g_0 g_0$	11001010110011101	2
IC	g_a	11011000101110001	χ_a (uniformly random index a)
IC	g_b	11000111111001110	χ_b (uniformly random index b)
$\circ(g_a, g_b)$	$g_a g_b$	10111100011100111	$\chi_a + \chi_b$

Table 2. Example simulation of the DLOG oracle with public and secret representations as in Table 1 and with a generic group of size $p = 29$. The vector $\vec{s}$ is sampled at random such that $D\vec{s} = \vec{r}$, and δ is the returned discrete logarithm as described in the DLOG oracle. For the vector $\vec{s}$ only the relevant entries are displayed as integers in $\mathbb{Z}_{29}$, and all other entries are marked with $*$, denoting a random integer which does not impact the computation of δ. The first DLOG query adds a zero-vector to D, since the DLOG (*i.e.*, 2) was already defined by the secret representation. Since D and $\vec{r}$ are empty, there is no restriction on the choice of $\vec{s}$. The 2nd and 3rd query return a random integer δ in the solution space. The response to the 4th query was already constrained by the 2nd and 3rd query, which can also be seen by checking that the vector corresponding to $\vec{b}$ was already in the row span of D before this query was received.

					Add to D			Add to $\vec{r}$
DLOG(g_v, g_w)	g_v	g_w	$\vec{s}$	δ	χ_1	χ_a	χ_b	
DLOG(g, g_1)	1	2	$(*, *, *)$	2 (forced by relations)	0	0	0	0
DLOG(g, g_a)	1	χ_a	$(*, 13, *)$	13 (random in $\mathbb{Z}_p$)	0	-1	0	-13
DLOG(g_a, g_c)	χ_a	$\chi_a + \chi_b$	$(*, 13, 4)$	$\frac{(13+4)}{13} \equiv 8 \mod 29$ (random in $\mathbb{Z}_p$)	0	7	-1	0
DLOG(g, g_b)	1	χ_b	$(*, 13, 4)$	4 (forced by relations)	0	0	-1	-4

For the other direction we know that there exists some $\vec{\omega}$ such that $\vec{\omega}^T D = \vec{b}^T$. Then we claim that the discrete logarithm between g and g_b is $b_0 + \vec{\omega}^T \vec{r}$. This is because we want $(b_0 + \vec{\omega}^T \vec{r}, \vec{0})$ to be $(D, \vec{r})$-equivalent to g_b, and indeed we can see that $-\vec{\omega}$ satisfies $-\vec{\omega}^T D = -\vec{b}^T$ and $-\vec{\omega}^T \vec{r} = b_0 - (b_0 + \vec{\omega}^T \vec{r})$ as desired. □

Table 1 gives an example of queries defining public and secret representations of the generic group model. Table 2 gives an example for simulating DLOG queries, showing the cases where the DLOG can take a uniformly random value in $\mathbb{Z}_p$ and when the DLOG between two elements is already defined (but has not yet been queried).

2.2 Security Model for Asymmetric PAKE

The BPR00 model [5] for security of an asymmetric password-authenticated key exchange protocol is defined by the interaction of a set of instances Π^i_P of principals P, which are either a client C or server S, and i denotes the i-th such instance. Each principal takes as input a long-lived (LL) secret. The client's LL secret is a password π; the server's secrets are the credentials $\mathit{cred}_S[C]$ that are established during a *Registration* phase. The model further defines a set of oracles that correspond to an adversary's interaction with principals that run the protocol in question. The adversary may receive a passive transcript (Execute queries), or actively engage (Send queries) in the communication. They may further request the session key (Reveal queries) or corrupt instances (Corrupt queries) which effectively returns the principal's long-lived keys (weak corruption). The security is defined by the adversary's probability to decide if they received a session key or a random string after submitting a Test query to a *fresh* instance. In the setting of quantum-annoying-ness, *fresh* means that neither the instance nor any partnered instance may be corrupted. A challenge bit that is sampled uniformly random before any interaction takes place decides which of the two (*i.e.*, real-or-random) is the case. In the generic group model, the adversary additionally gets access to the group operation and discrete logarithm oracle (cf. Section 2.1). A protocol is quantum-annoying in the BPR00 model, if the adversary's advantage to output the challenge bit is bounded by the number of Send queries (q_{Send}) and discrete logarithm queries (q_{Dlog}),

$$\mathrm{Adv}^{\text{QA-BPR}}_{\text{Protocol}}(\mathcal{A}) = \left| \Pr\left[\mathcal{A} \text{ guesses challenge bit}\right] - \frac{1}{2} \right| \leq \frac{q_{\text{Send}} + q_{\text{Dlog}}}{N} + \epsilon\,, \quad (2)$$

with a password space of size N and ϵ negligible in the security parameter κ.

2.3 KHAPE-HMQV

The KHAPE compiler [10] transforms a key-hiding authenticated key exchange, a PAKE, a random oracle, and an ideal cipher into an asymmetric PAKE which provides key establishment with key integrity and confirmation, mutual authentication and forward secrecy. A highly efficient instantiation [10, Fig. 14] uses the HMQV [14] protocol, the security of which is based on the computational Diffie–Hellman problem.

KHAPE is split into a *registration* and an *aPAKE* phase. During registration the server generates the KH-AKE key-pairs $(a, A := g^a)$, $(b, B := g^b)$, partially encrypts them using the password as a key, $e \leftarrow \mathrm{IC}.E(\pi, a, B)$, and stores the ciphertext along with (A, b). All other values are discarded. In the *aPAKE* phase the server generates a key-pair (y, Y) and sends (Y, e) to the client. The client decrypts e using their password and generates a key pair (x, X). A Diffie–Hellman session is computed from (a, x, B, Y) which is used to derive a key-confirmation value τ, and later the session key. The key confirmation is sent along with the value X to the server, who computes the equivalent Diffie–Hellman session from (b, y, A, X), verifies the key confirmation, and either computes a session key and

Client	Message	Server
Registration on Server input (π, C)		$(a, A), (b, B)$ fresh AKE keys; $\mathsf{sk} \xleftarrow{\$} \{0,1\}^\kappa$
		$e \leftarrow \mathrm{IC}_1.E(\pi, a, B, \mathsf{sk})$
		store $\mathit{creds}_S[C] = (b, A, e, \mathsf{sk})$
aPAKE $\boxed{C(\mathrm{sid}, S, \pi)}$		$\boxed{S(\mathrm{sid}, C, \mathit{creds}_S[C])}$
$a, B, \mathsf{sk} \leftarrow \mathrm{IC}_1.D(\pi, e)$	$\xleftarrow{e, Y}$	$y \xleftarrow{\$} \mathbb{Z}_p,\ Y \leftarrow g^y$
$x \xleftarrow{\$} \mathbb{Z}_p,\ X \leftarrow g^x$		
$h_X \leftarrow \mathcal{H}_1(\mathrm{sid}, C, S, X),\ h_Y \leftarrow \mathcal{H}_1(\mathrm{sid}, C, S, Y)$		
$\sigma_C \leftarrow \left(Y \cdot B^{h_Y}\right)^{x + h_X \cdot a}$		
$k_1 \leftarrow \mathcal{H}_2(\mathrm{sid}, C, S, X, Y, \sigma_C)$		
$c_X \leftarrow \mathrm{IC}_2.E(\mathsf{sk}, X)$	$\xrightarrow{c_X, \tau}$	$\triangleright$ In KHAPE-HMQV, X is sent instead of c_X
$\tau \leftarrow \mathit{prf}(k_1, 1)$		
		$X \leftarrow \mathrm{IC}_2.D(\mathsf{sk}, c_X)$
		$h_X \leftarrow \mathcal{H}_1(\mathrm{sid}, C, S, X), h_Y \leftarrow \mathcal{H}_1(\mathrm{sid}, C, S, Y)$
		$\sigma_S \leftarrow \left(X \cdot A^{h_X}\right)^{y + h_Y \cdot b}$
		$k_2 \leftarrow \mathcal{H}_2(\mathrm{sid}, C, S, X, Y, \sigma_S)$
		if $\tau \neq \mathit{prf}(k_2, 1)$**:** $\gamma = \mathrm{K}_2 = \perp$
if $\gamma \neq \mathit{prf}(k_1, 2)$**:** $K_1 \leftarrow \perp$	$\xleftarrow{\gamma}$	**else:** $\gamma \leftarrow \mathit{prf}(k_2, 2),\ \mathrm{K}_2 \leftarrow \mathit{prf}(k_2, 0)$
else: $K_1 \leftarrow \mathit{prf}(k_1, 0)$		

Figure 1. QA-KHAPE: quantum-annoying variant of KHAPE-HMQV [10, Fig 14], with our changes compared to KHAPE-HMQV highlighted.

a new key confirmation (which is send to the client), or sets both to $\perp$. The client checks the key confirmation and computes the session key, or sets it to $\perp$.

In the quantum-annoying setting, KHAPE-HMQV is susceptible to an offline attack on the password using a single discrete logarithm query. Given a transcript (e, Y, X, τ) an adversary can determine a list of possible values for KH-AKE key-pairs: each password guess π_i gives a pair of candidate values $(a_i, B_i) \leftarrow \mathrm{IC}.D(\pi_i, e)$. Additionally, they can query the discrete logarithm oracle once on the value X, receiving x. Then for each password guess (*i.e.*, for each a_i, B_i), they can verify if the Diffie–Hellman completion results in the key-confirmation value τ from the transcript, effectively providing an offline method to check passwords.

2.4 Quantum-Annoying KHAPE-HMQV

Our QA-KHAPE protocol, presented in Fig. 1, is a quantum-annoying aPAKE. The construction is based on KHAPE-HMQV and requires only minimal changes, which are highlighted in the figure. During the *registration* phase the server generates an additional secret key sk which is then encrypted using the π and stored as part of the credentials. Correspondingly, during the *aPAKE* phase the client decrypts e obtaining this key sk, which they use to encrypt the ephemeral value X, resulting in the ciphertext c, which is then sent to the server. Briefly speaking, QA-KHAPE is quantum-annoying because an adversary receiving a transcript must now solve a discrete logarithm for *every* decryption of c or e to verify if a password guess was correct. This comes at the cost of an additional secret key to be stored as credentials, which increases the size of first message from server to client. The client has to perform one additional decryption and encryption, while the server has to perform one additional decryption.

Security. The QA-KHAPE protocol is a quantum-annoying aPAKE in the generic group (cf. Section 2.1), ideal cipher and random oracle model and features mutual authentication and key confirmation. No perfect forward secrecy can be achieved in the setting of quantum-annoying for KHAPE-HMQV, because compromise of any party releases a static secret that, together with the public value e, removes all the ambiguity on the group elements in question (*i.e.*, A, B, X). This enables an offline attack on the password using only a single discrete logarithm query. Note that a quantum-annoying PAKE achieving perfect forward secrecy would mean to establish a secure, authenticated key without taking advantage of the password or credentials, which seemingly contradicts the main point of a PAKE; establishing this formally is an interesting question for future work.

All other properties of KHAPE-HMQV are preserved, for example, security based on the computational Diffie–Hellman assumption against purely classical attackers, and thus a full fall back to security of KHAPE-HMQV. The quantum-annoying security is summarized in our main contribution, Theorem 1.

Theorem 1. *Let $\mathbb{G}$ be a cyclic group of size p, $\mathcal{H}_1, \mathcal{H}_2$ be random oracles and IC_1, IC_2 ideal ciphers with ciphertext space $\{0,1\}^{n_1}$, $\{0,1\}^{n_2}$ respectively. Let $q_{\text{SEND}}, q_{\text{EXEC}}, q_{\mathcal{H}_i}, q_{IC_i}, q_\circ, q_{\text{DLOG}}$ be the number of queries to the QA-BPR oracles, and let ϵ_{prf} an adversary's chance to distinguish prf from a random function. Let N be the size of the password space for π. Then the advantage of an adversary to win the QA-BPR game for the QA-KHAPE protocol in Fig. 1 is bounded by*

$$Adv^{QA\text{-}KHAPE}_{QA\text{-}BPR} \leq \frac{q_{\text{DLOG}} + q_{\text{SEND}}}{N} + \epsilon \tag{3}$$

$$\epsilon := \frac{q_{\text{EXEC}} + q_{\text{SEND}}}{\epsilon_{prf}^{-1}} + \frac{(q_{IC_1} + q_{IC_2} + q_\circ)^2 + (q_{\text{DLOG}} q_\circ^2)}{p} + \frac{q_{\text{EXEC}}}{2^{n_1}} + \frac{q_{\text{EXEC}} + q_{\text{SEND}}}{2^{n_2}}$$

$$+ \frac{q_{\text{SEND}} \cdot (q_\circ + 1)}{p} + \frac{(2q_{IC_1} + q_{IC_2})}{p} + \frac{(q_{IC_1})}{2^\kappa} + \frac{(2q_{IC_1}^2 + q_{IC_2}^2)}{p} + \frac{(q_{IC_1}^2)}{2^\kappa} + \frac{q_{\mathcal{H}_2}}{p}$$

We prove Theorem 1 in two steps: first, in Section 3.1, we introduce the KHAPE$_{\text{CORE}}$-game that captures the quantum-annoying property of QA-KHAPE in the generic group model. Briefly speaking, the game models the aPAKE without key-confirmation values and is defined such that any adversary can only win if they query the *correct* Diffie–Hellman completion to the random oracle. This allows us to quantify the number of discrete logarithm queries required, and to prove that every password guess requires either an online interaction, or a respective discrete logarithm query. Formally, this is captured in Theorem 2 which we prove in Section 3.2. Second, we reduce the QA-BPR-security of the QA-KHAPE protocol to the $\text{KHAPE}_{\text{CORE}}$-game, which is represented by Theorem 1 and which we prove in Section 4. Together, these yield the proof of the quantum-annoying property.

3 Generic Group Security: $\text{KHAPE}_{\text{CORE}}$

We define a game $\text{KHAPE}_{\text{CORE}}$ that captures the quantum annoying property of the protocol in Fig. 1, namely the indistinguishability of the keys k_1, k_2 from random, which translates the approach of [7, Sec 3] into the setting of an aPAKE.

3.1 Security Game

The game is defined over a set $[L]$ registrants; each $l \in [L]$ is associated with static, secret variables $\pi_l, \text{sk}_l, a_l, B_l$ and a static, public variable e_l. The variables are set on initialization of the $\text{KHAPE}_{\text{CORE}}$-game via the Registration oracle (cf. Algorithm 1), along with uniformly random sampled challenge bit s. Additionally, each registrant l is associated with a counter ctr_l initialized to 0 corresponding to the interaction with the lth set of static variables. Each interaction is called an instance. The adversary may interact with an arbitrary number of registrants and instances through a set of oracles, eventually allowing the adversary to obtain the keys k_1, k_2. The challenge bit determines if these keys are real (if $s = 0$), in which case they are computed from Diffie-Hellman session, or random (if $s = 1$).

Interface. The oracles take as input a value l matching a set of static variables which are used by the game to respond to a query. Ephemeral variables for an instance (l, ctr_l) are stored for consistent use by the other oracles. The PassiveExec oracle (cf. Algorithm 2) corresponds to a passive execution of the protocol in Fig. 1, excluding the key confirmation values. The Active_C or Active_S oracles (cf. Algorithms 4 and 5), correspond to interacting with, or impersonating, either party in the QA-KHAPE protocol, and thus at most one of the two may be queried for each instance. The Active oracles compute, depending on the value of the challenge bit s, either a key value $k_{l,ctr_l,i}$ from the input and the static variables or output a uniformly random string. The GetStatic oracle (cf. Algorithm 3) mimics the corruption of parties, which causes the game to reprogram the outputs of the Active oracles into the respective positions before it returns the secret static variables. Finally, the adversary is given access to the random oracles $\mathcal{H}_1, \mathcal{H}_2$, the block-ciphers IC_1, IC_2 modeled by ideal ciphers and access to an interface of the generic group model.

Output. The $\text{KHAPE}_{\text{CORE}}$-game outputs 1 if the adversary's output matches the challenge bit s or if they if they query $\mathcal{H}_2(l, m, X, Y, \sigma_{l,C})$ (respectively $\mathcal{H}_2(l, m, X, Y, \sigma_{l,S})$) after submitting a query $\text{Active}_C(l, e, Y)$ (respectively

Algorithm 1 REGISTRATION(l)

```
1: $\pi_l \xleftarrow{\$} [N]$, $\mathrm{sk}_l \xleftarrow{\$} \{0,1\}^\kappa$
2: $a_l, b_l \xleftarrow{\$} \mathbb{Z}_p$
3: $B_l \leftarrow g^{b_l}$, $A_l \leftarrow g^{a_l}$
4: $e_l \leftarrow \mathrm{IC}_1.E(\pi_l, a_l, B_l, \mathrm{sk}_l)$
5: Store $\pi_l, \mathrm{sk}_l, e_l, A_l, b_l$
```

Algorithm 2 PASSIVEEXEC(l)

```
Require: $l \leq L$
1: Get stored $\pi_l, \mathrm{sk}_l, e_l, A_l, b_l$
2: Increment $ctr_l$
3: $x_{l,ctr_l}, y_{l,ctr_l} \xleftarrow{\$} \mathbb{Z}_p$
4: $X_{l,ctr_l} \leftarrow g^{x_{l,ctr_l}}$, $Y_{l,ctr_l} \leftarrow g^{y_{l,ctr_l}}$
5: $c_{l,ctr_l} \leftarrow \mathrm{IC}_2.E(\pi_l, X_{l,ctr_l})$
6: Store $Y_{l,ctr_l}, ctxt_{l,ctr_l}$
7: return $e_l, Y_{l,ctr_l}, c_{l,ctr_l}$
```

Algorithm 3 GETSTATIC(l)

```
Require: $l \leq L$
1: Mark $l$ corrupted; Get stored $\pi_l, \mathrm{sk}_l$
2: for $m = 0, \dots, ctr_l$ do
3:     for $k_{l,m}, c_{l,m} \leftarrow \mathrm{ACTIVE}_C(l, e, Y)$
4:         $a, B, \mathrm{sk} \leftarrow \mathrm{IC}_1.D(\pi_l, e)$
5:         $h_X = \mathcal{H}_1(l, m, X_{l,m})$, $h_Y = \mathcal{H}_1(l, m, Y)$
6:         $\sigma_C \leftarrow (Y \circ B^{h_Y})^{x_{l,m} + h_X \cdot a}$
7:         $\mathcal{H}_2(l, m, X_{l,m}, Y, \sigma_C) := k_{l,m,1}$
8:     for $k_{l,2} \leftarrow \mathrm{ACTIVE}_S(l, c)$
9:         Get stored $Y_{l,m}$
10:        $X \leftarrow \mathrm{IC}_2.D(\mathrm{sk}_l, c)$
11:        $h_X = \mathcal{H}_1(l, m, X)$, $h_Y = \mathcal{H}_1(l, m, Y_{l,m})$
12:        $\sigma_S \leftarrow (X \circ A_{l,m}^{h_X})^{y_{l,m} + h_Y \cdot b_{l,m}}$
13:        $\mathcal{H}_2(l, m, X, Y_{l,m}, \sigma_S) := k_{l,m,2}$
14: return $(\pi_l, \mathrm{sk}_l)$
```

ACTIVE$_S(l, c)$), but before querying GETSTATIC(l) on the instance. The adversary is then said to *win* the game. The restriction on the GETSTATIC oracle mimics the fact that forward secrecy cannot be achieved in the quantum annoying model. The conditions under which the game outputs 1 are analogous to the winning conditions of [7, Sec. 3.1].

Theorem 2 (Security of KHAPE$_{\text{CORE}}$). *Let q_{AE_C}, q_{AE_S} be the number of queries to the* ACTIVE *and q_{PE} the number of queries to the* PASSIVEEXEC *oracle. Let $q_{IC_i}, q_{\mathcal{H}_i}, q_\circ, q_{\mathrm{DLOG}}$ be the number of queries to the ideal cipher, random oracle, group operation and discrete logarithm oracles respectively. Then $\mathcal{A}$'s probability to win the KHAPE$_{CORE}$-game is bounded by*

$$\Pr\left[KHAPE_{CORE} \Rightarrow 1\right] \leq \frac{1}{2} + \frac{q_{\mathrm{AE}_C} + q_{\mathrm{AE}_S} + q_{\mathrm{DLOG}}}{N} + \epsilon_{CORE} \tag{4}$$

$$\epsilon_{CORE} := \overbrace{\frac{(q_{IC_1} + q_{IC_2} + q_\circ)^2 + (q_{\mathrm{DLOG}} q_\circ^2)}{p}}^{//\, G_0 \rightsquigarrow G_1} + \overbrace{\frac{(q_{\mathrm{AE}_C} + q_{\mathrm{AE}_S}) \cdot (q_\circ + 1)}{p}}^{//\, G_3 \rightsquigarrow G_4}$$
$$+ \underbrace{\frac{q_{\mathrm{PE}}}{2^{n_1}} + \frac{q_{\mathrm{PE}} + q_{\mathrm{AE}}}{2^{n_2}}}_{//\, G_2 \rightsquigarrow G_3} + \underbrace{\frac{(2q_{IC_1} + q_{IC_2})}{p} + \frac{(q_{IC_1})}{2^\kappa} + \frac{(2q_{IC_1}^2 + q_{IC_2}^2)}{p} + \frac{(q_{IC_1}^2)}{2^\kappa}}_{//\, G_1 \rightsquigarrow G_2} . \tag{5}$$

3.2 Proof of Theorem 2

The proof of Theorem 2 shows, informally, that the adversary's chance to win the KHAPE$_{\text{CORE}}$-game is limited by their ability to query the DLOG oracle

Algorithm 4 $\textsc{Active}_C(l, e, Y)$

```
Require: $l \leq L$
1: Increment $ctr_l$; Get stored $\pi_l$
2: $a, B, \mathrm{sk} \leftarrow \mathrm{IC}_1.D(\pi_l, e)$
3: $x_{l,ctr_l} \xleftarrow{\$} \mathbb{Z}_p$, $X_{l,ctr_l} \leftarrow g^{x_{l,ctr_l}}$
4: $c_{l,ctr_l} \leftarrow \mathrm{IC}_2.E(\mathrm{sk}, X_{l,ctr_l})$
5: if challenge $s = 0$ or $l$ corrupted :
6:     $h_X = \mathcal{H}_1(l, ctr_l, X_{l,ctr_l})$, $h_Y = \mathcal{H}_1(l, ctr_l, Y)$
7:     $\sigma_C \leftarrow (Y \circ B^{h_Y})^{x_{l,ctr_l} + h_X \cdot a}$
8:     $k_{l,ctr_l,1} = \mathcal{H}_2(l, ctr_l, X_{l,ctr_l}, Y, \sigma_C)$
9: else $k_{l,ctr_l,1} \leftarrow \{0,1\}^n$
10: return $k_{l,ctr_l,1}, c_{l,ctr_l}$
```

Algorithm 5 $\textsc{Active}_S(l, c)$

```
Require: $l \leq L$
1: Increment $ctr_l$; Get stored $\mathrm{sk}_l$
2: if challenge $s = 0$ or $l$ corrupted :
3:     $X \leftarrow \mathrm{IC}_2.D(\mathrm{sk}_l, c)$
4:     $h_X = \mathcal{H}_1(l, ctr_l, X)$, $h_Y = \mathcal{H}_1(l, ctr_l, Y_{l,ctr_l})$
5:     $\sigma_C \leftarrow (X \circ A_l^{h_X})^{y_{l,ctr_l} + h_Y \cdot b_{l,ctr_l}}$
6:     $k_{l,ctr_l,2} = \mathcal{H}_2(l, ctr_l, X, Y_{l,ctr_l}, \sigma_{l,C})$
7: else $k_{l,ctr_l,2} \leftarrow \{0,1\}^n$
8: return $k_{l,ctr_l,2}$
```

on the *correct* group element, or any of the $\textsc{Active}$ oracles on a ciphertext encoding a group element the discrete logarithm of which is known to them. In the $\mathrm{KHAPE}_{\mathrm{CORE}}$-game, the group elements in question are computed as

$$\begin{aligned} \sigma_C &= \left(Y \cdot B^{h_Y}\right)^{x + h_X \cdot a} = Y^x \cdot Y^{h_x \cdot a} \cdot B^{h_Y \cdot x} \cdot B^{h_y \cdot h_x \cdot a} \\ &= g^{xy} \cdot g^{h_X \cdot a \cdot y} \cdot g^{h_Y \cdot b \cdot x} \cdot g^{h_Y \cdot h_X \cdot a \cdot b} \\ &= X^y \cdot A^{h_X \cdot y} \cdot X^{h_Y \cdot b} \cdot A^{h_X \cdot h_Y \cdot b} = \left(X \cdot A^{h_X}\right)^{y + h_Y \cdot b} = \sigma_S \,, \end{aligned} \tag{6}$$

where computing σ_C, σ_S depends on either the knowledge of $\textsc{Dlog}(g, B)$ or $\textsc{Dlog}(g, X)$. The framework presented in Section 2.1, allows us to quantify if these element are knowable based on the number of discrete logarithm queries. This is possible, because the relevant group elements X, B are encrypted under the ideal cipher. On a decryption query the ideal cipher can return a public representations that does not admit a relation to a previously received group element known by the adversary. To learn any such relation, the adversary then has to query the $\textsc{Dlog}$ oracle. Specifically, the *relevant* group elements $\{B_{l,i}, X_{l,i}\}_{i \in [N]}$ correspond to decryptions of (e, c) using a password guess π_i and sk_i as keys respectively. In the $\mathrm{KHAPE}_{\mathrm{CORE}}$-game, the *correct* pair $B_{l,i}, X_{l,i}$ is chosen during the *Registration* phase and in the $\textsc{Active}$ oracles. Due to the values being encrypted by the ideal ciphers, the simulation does not need to commit to any actual pair B_i, X_i.

We prove this by presenting a sequence of game hops where the the initial game G_0 (cf. Section 3.2) is the $\mathrm{KHAPE}_{\mathrm{CORE}}$-game as defined in Section 3.1, and G_4 (cf. Section 3.2) is modified such that the keys k_1, k_2 are chosen uniformly random for every instance, and where the discrete logarithm of g and the group elements B, X remain undefined unless sufficiently constrained by queries to the $\textsc{Dlog}$ and $\textsc{Active}$ oracles. They are undefined because the ciphertexts (e, c) are indistinguishable from random strings, and the key pair (π, sk) is no longer defined from the $\textsc{PassiveExec}$ or $\textsc{Active}$ oracles. That means that the *correct* values for (B, X) may correspond to any of the N possible pairs. As long as there is a degree of freedom left for these representations, the discrete logarithm relative to g is also not defined, and the random oracle cannot be queried on the respective Diffie–Hellman completion. These are only defined either if an instance is corrupted, or if sufficiently many discrete logarithms have been queried, allowing to quantify the adversary's probability to win relative to the number of DLOG queries.

G_0 (KHAPE$_{CORE}$-game). This is the KHAPE$_{\text{CORE}}$ as described in Section 3.1.

G_1 (GGM). We modify the responses to the group operation $\circ$ and DLOG oracle by simulating the generic group as described in Section 2.1. The generator initially given to the adversary is $g_1 = g$, which corresponds to the secret representation 1. The secret representation of the neutral element is 0. Recall that the password space is of size N. The secret variables are represented as a set $\{\chi_{l,i}, \chi_{l,i}\}_{i\in[N]}$ corresponding to the pairs $B_{l,i}, X_{l,i}$ that can be obtained when querying the ideal ciphers on possible values for π_l or sk_l. The ideal cipher IC_i is maintained via a table T_{IC_i}. On query $\text{IC}_1.D(\pi, e)$, if $T_{\text{IC}_1}[\pi, e]$ is defined, return $T_{\text{IC}_1}[\pi, e]$. Otherwise, sample a random index $j \xleftarrow{\$} [N]$ for the secret representation and a public representation $g_V \xleftarrow{\$} \{0,1\}^n$, both of which are added to the table $T_{\text{ggm}}[\chi_{i,j}] := g_V$; The public representation g_V is returned. The simulation of IC_2 is analog.

The modification changes the distribution of the group elements: public representations returned from the ideal ciphers (on new inputs) in the simulation are unique, whereas the adversary would expect a collision after $\sqrt{p}$ new queries. Additionally, the adversary would expect to see collisions between random public representations, and the elements returned from (sufficiently many) group operations. This happens with probability $(q_{\text{IC}_1} + q_{\text{IC}_2} + q_\circ)^2/p$.

Further, a group element may be assigned two distinct public representations, if first computed from group operations and then returned from an IC query (or vice versa). For example, if the public representation g_x was returned from an IC query, and the representation $g_{\bar{x}} = g^x$ was assigned from group operations, then the adversary may detect the modification by computing $\text{DLOG}(g_1, g_x) = x$. The probability that this happens for group elements randomly assigned by the ideal cipher and for all DLOG queries is $q_{\text{DLOG}} q_\circ^2/p$. Overall, the adversary can distinguish the two games with probability at most

$$\frac{(q_{\text{IC}_1} + q_{\text{IC}_2} + q_\circ)^2 + q_{\text{DLOG}} q_\circ^2}{p} . \tag{7}$$

G_2 (Ideal Ciphers Output). We change the ideal ciphers to output unique, random values when queried on a new input. On query $\text{IC}_1.D(\pi, e)$, if $T_{\text{IC}_1}[\pi, e]$ is not defined, the ideal cipher IC_1 samples key pairs $a, b \xleftarrow{\$} \mathbb{Z}_p$ and $\text{sk} \leftarrow \{0,1\}^\kappa$, generates public keys $A = g^a$, $B = g^b$ and a key $\text{sk} \leftarrow \{0,1\}^\kappa$, and programs $T_{\text{IC}_1}[\pi, e] := a, B, \text{sk}$. In the case of a collision, *i.e.*, if (a, B, sk) has been assigned to a value in the map $T_{\text{IC}_1}[\pi, \cdot]$ for any value $\cdot$, G_2 aborts. Since (a, B, sk)

are independent random variables, the probability for an abort is bounded by $2q_{\mathrm{IC}_1}/p + q_{\mathrm{IC}_1}/2^\kappa$, neglecting the a deduction for a simultaneous collision of all variables. Since the values a, B, sk are *unique*, two different queries will never output the same values, whereas the adversary would eventually expect a collision in G_1. The same argument applies to IC_2. In total, the divergence is bounded by

$$\frac{2q_{\mathrm{IC}_1} + q_{\mathrm{IC}_2}}{p} + \frac{(q_{\mathrm{IC}_1})}{2^\kappa} + \frac{2q_{\mathrm{IC}_1}^2 + q_{\mathrm{IC}_2}^2}{p} + \frac{(q_{\mathrm{IC}_1}^2)}{2^\kappa}\,. \tag{8}$$

G_3 (Random Ciphertexts). We modify the game to not sample any keys π and sk and to output random strings $e \xleftarrow{\$} \{0,1\}^{n_1}$, $c \xleftarrow{\$} \{0,1\}^{n_2}$ in the PassiveExec and Active$_C$ oracles, which removes the game's commitment to any value stored in (e, c). Analogous to the modification in G_2, the game aborts if the values were previously assigned. At the same time the GetStatic oracle is changed to reflect the modification: the simulation first decrypts the ciphertext using freshly sampled keys π and sk. The Dlog oracle provides the values necessary to compute the Diffie–Hellman session such that the output of the Active oracles can be programmed into the correct position of the ideal cipher. For a detailed algorithm of the modified (and "final") PassiveExec and GetStatic oracle we refer the reader to Appendix A1. The distribution of (e, c) returned by PassiveExec and Active$_C$ is the same as in G_2 unless it aborts. Since (e, c) are sampled uniformly random from the ciphertext space, the probability for this to happen is bounded by

$$\frac{q_{\mathrm{PE}}}{2^{n_1}} + \frac{q_{\mathrm{PE}} + q_{\mathrm{AE}_C}}{2^{n_2}}\,. \tag{9}$$

G_4 (Embed Random Keys). The Active oracles are modified to always return random strings $k \xleftarrow{\$} \{0,1\}^\kappa$ for non-corrupted instances. To notice this change, the adversary must query $\mathcal{H}_2(ctr, X, Y, \sigma_i)$, where the Diffie–Hellman completion σ_i depends on either the knowledge of $\mathrm{Dlog}(g, X)$ and B, or the knowledge of $\mathrm{Dlog}(g, B)$ and X, both of which are not defined by the game unless GetStatic has been queried, in which case the adversary cannot win the game anymore.

The probability that $\mathrm{Dlog}(g, X)$ or $\mathrm{Dlog}(g, B)$ are knowable to the adversary is bounded by Corollary 1, which tells us that the discrete logarithms are defined if an only if $\vec{b}, \vec{x}$ are in the row span of D. Both, $\vec{b}$ and $\vec{x}$, are basis vectors with a 1 at the position of the random index associated with the respective secret variable. The number of basis vectors that can appear in the row span are upper bounded by the rank of the matrix D, which is increased by 1 for each Dlog query. Therefore, the probability that the adversary can force the definition for any one value out of N many of these is bounded by q_{Dlog}/N.

Remark: Only public representations returned from the ideal ciphers, and possibly group elements that come from group operation applied to these group elements, provide *useful* input to the Dlog oracle, since the discrete logarithm relation for group elements originating purely from g is already known to the adversary. Therefore, the probability is

$$\frac{\min(q_{\mathrm{IC}_1} + q_{\mathrm{IC}_2}, q_{\mathrm{Dlog}})}{N} \le \frac{q_{\mathrm{Dlog}}}{N}\,. \tag{10}$$

Additionally, the adversary may submit a query with a group element, the discrete logarithm of which is known to them. The input $\hat{e}$ to the $\textsc{Active}_C$ oracle is either a value formerly returned from a previous query to $\textsc{PassiveExec}$, in which case the adversary must also query the ideal cipher and the $\textsc{Dlog}$ oracle and there is a chance of $(q_\circ+1)/p$ that the discrete logarithm of the group element decrypted by $\textsc{Active}_C$ is known to them. If $\hat{e}$ was crafted by the adversary, *i.e.*, if they queried the ideal cipher on values $\hat{a}, \hat{B}, \hat{\mathsf{sk}}$ such that the discrete logarithm of $\hat{B}$ is known to them, then they expect an $1/N$ chance that there choice of $\hat{pw}$ was correct, and that $\textsc{Active}_C$ used $\hat{b}$ to compute the Diffie–Hellman session.

In total, this result in a divergence for $\textsc{Active}_C$ queries bounded by

$$\frac{q_{\mathrm{AE}_C} \cdot (q_\circ + 1)}{p} + \frac{q_{\mathrm{AE}_C}}{N}. \tag{11}$$

For $\textsc{Active}_S$, the adversary may submit a value $\hat{c_x}$ for which the same arguments hold, resulting in a total probability for either of both occurring of

$$\frac{(q_{\mathrm{AE}_C} + q_{\mathrm{AE}_S}) \cdot (q_\circ + 1)}{p} + \frac{(q_{\mathrm{AE}_C} + q_{\mathrm{AE}_S})}{N}. \tag{12}$$

Finally, the adversary's advantage to distinguish the simulation from the real game based on $\textsc{Dlog}$ queries depends on the knowledge of at least one key from a $\textsc{Active}$ oracle, resulting in a factor of $\min(q_{\mathrm{AE}_C} + q_{\mathrm{AE}_S}, 1)$, thus bounding the overall divergence by

$$\min(q_{\mathrm{AE}_C} + q_{\mathrm{AE}_S}, 1)\cdot \left(\frac{(q_{\mathrm{AE}_C} + q_{\mathrm{AE}_S}) \cdot (q_\circ + 1)}{p} + \frac{(q_{\mathrm{AE}_C} + q_{\mathrm{AE}_S} + \min(q_{\mathrm{IC}_1} + q_{\mathrm{IC}_2}, q_{\textsc{Dlog}}))}{N}\right) \tag{13}$$

$$\leq \frac{(q_{\mathrm{AE}_C} + q_{\mathrm{AE}_S}) \cdot (q_\circ + 1)}{p} + \frac{(q_{\mathrm{AE}_C} + q_{\mathrm{AE}_S} + q_{\textsc{Dlog}})}{N} \tag{14}$$

.

In G_4, the $\textsc{PassiveExec}$ and $\textsc{Active}_C$ oracle output random values as ciphertexts e, c that do not commit to any values a, B, π or X. Particularly, the values $\textsc{Dlog}(g, X), \textsc{Dlog}(g, B)$ are defined only upon corruption or after a number of $\textsc{Active}$ and $\textsc{Dlog}$ queries relative to the password space N. The $\textsc{Active}_*$ oracles further output a random key independent of the challenge bit $s = 0$. The adversary is left with either guessing the challenge bit, or querying values to $\mathcal{H}_2$. This concludes the proof of Theorem 2. □

4 aPAKE Security: Sketch of Proof of Theorem 1

The security of the QA-KHAPE protocol (cf. Figure 1) is proven in the QA-BPR (cf. Section 2.2) model. Recall that the adversary may interact through the $\textsc{Execute}$, $\textsc{Send}$, $\textsc{Reveal}$, $\textsc{Corrupt}$ and $\textsc{Test}$ oracles after the *Registration* phase, where the protocol defines how the principals respond. Additionally, the adversary has access to the group operation, DLOG and random oracle, ideal cipher and pseudo-random function prf, as described in Section 2.1, and is bounded by Theorem 1. In this section we offer a summary of the proof's main ideas and provide a complete description in Appendix A2.

We consider a sequence of games starting with G_0, which corresponds to the QA-KHAPE protocol illustrated in Figure 1. As we progress to G_3, the sessions keys are chosen independent and uniformly at random, ensuring that the adversary $\mathcal{A}$ is reduced to a simple guessing attack. Throughout this reduction process, we present a an adversary $\mathcal{B}$ on the KHAPE$_{\text{CORE}}$-game, which maintains a mapping between instances of the KHAPE$_{\text{CORE}}$-game and instances of principals in the QA-BPR-model. To achieve this, we utilize a procedure called *CoreMap*, which bears resemblance to the GETUV procedure described in [7, App. B.2] (cf. Appendix A1 for further details). Intuitively, two counters $ctr_{C,S}$ and $ctr_{C,S,\text{sid}}$, are employed to either map to a set of static variables indexed by l or to an instance ctr_l in the KHAPE$_{\text{CORE}}$-game. The oracles provided by the KHAPE$_{\text{CORE}}$ challenger are referred to as KHAPE$_{\text{CORE}}$.Oracle.

In the sequence of games, the first modification occurs in G_1, where we replace the keys k_1, k_2 in the EXECUTE oracle with random strings. This change is reflected in the CORRUPT oracle, which now programs the random string instead of the actual keys into the random oracle. The distinction between G_0 and G_1 can be reduced to an attacker on the KHAPE$_{\text{CORE}}$-game. To differentiate between these two games, we can provide an extractor for a winning query: utilizing the *CoreMap*, the calls to the EXECUTE and SEND oracles can be forwarded to the KHAPE$_{\text{CORE}}$.PASSIVEEXEC, which returns $(e_{C,S,sid}, Y_{C,S,sid}, c_{C,S,sid})$, These outputs can be queries to KHAPE$_{\text{CORE}}$.ACTIVE$_*$, which provides $(c_{C,S,sid}, k_{C,S,1})$ or $k_{C,S,2}$. Using from these keys, the confirmation values (τ, γ) that genuinely follow the protocol are computed.

In the modified game, all calls to the ideal cipher are simulated perfectly by forwarding queries to, and responses from, the KHAPE$_{\text{CORE}}$-game. However, queries to the random oracle are not simulated perfectly, because the adversary can submit a query not associated with an instance in the KHAPE$_{\text{CORE}}$, and that gets mapped to different value in the KHAPE$_{\text{CORE}}$-game later on. Nonetheless, in every call to $\mathcal{H}_2$ in the KHAPE$_{\text{CORE}}$, either the group element X or Y is chosen uniformly at random from the group, ensuring that the the adversary's probability of querying the same group element beforehand is at most $q_{\mathcal{H}_2}/p$.

The TEST queries in G_1 are simulated perfectly, since the keys are chosen uniformly at random, ensuring that the key confirmation values follow the expected distribution. In the non-TEST queries, as the KHAPE$_{\text{CORE}}$ instances are real-or-random based on the respective challenge bit. Specifically, the keys (k_1, k_2) are real-or-random values. However, the adversary can only detect this if they can query the random oracle $\mathcal{H}_2$ for the position of the programmed random keys. Such a query would immediately be a winning query in the KHAPE$_{\text{CORE}}$-game. Therefore, the adversary's ability to distinguish between the two games can be limited by their capability to win the KHAPE$_{\text{CORE}}$-game, which can be quantified relative to the number of DLOG queries they submit.

In game G_2, the modification is extended to the active queries (*i.e.*, Send). The extractor for the KHAPE$_{\text{CORE}}$-game acts nearly identical. The only difference is that the quantification of KHAPE$_{\text{CORE}}$ advantage is now also impacted by the adversary's ability to query ciphertexts (e, c), the discrete logarithm of the decryption of which is knowable to them. This adds an additional term for the number of Send queries. The probability to detect the modifications from the non-Test queries in game G_1 and G_2 result in the term $(q_{\text{DLog}}+q_{\text{Send}})/N + \epsilon_{\text{CORE}}$.

The last modification occurs in game G_3, where we exchange the sessions keys with random values. Since the keys (k_1, k_2) are already random strings, and the session key is the output of the *prf*, this can be reduced to the adversary's ability to distinguish *prf* from a random function, *i.e.*, ϵ_{prf}. This adds an additional divergence of $(q_{\text{Execute}} + q_{\text{Send}})\epsilon_{prf}$ and concludes the proof.

Acknowledgements M.T. was supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs. D.S. was supported by Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery grant RGPIN-2022-03187 and NSERC Alliance grant ALLRP 578463-22.

References

1. IEEE standard specification for password-based public-key cryptographic techniques. IEEE Std 1363.2-2008 (2009). https://doi.org/10.1109/IEEESTD.2009.4773330
2. Information technology — personal identification — ISO-compliant driving licence. ISO/IEC 18013-3:2027 (2017)
3. Abdalla, M., Haase, B., Hesse, J.: Security analysis of CPace. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021, Part IV. LNCS, vol. 13093, pp. 711–741. Springer, Heidelberg (Dec 2021). https://doi.org/10.1007/978-3-030-92068-5_24
4. Beguinet, H., Chevalier, C., Pointcheval, D., Ricosset, T., Rossi, M.: Get a cake: Generic transformations from key encaspulation mechanisms to password authenticated key exchanges. In: Applied Cryptography and Network Security: 21st International Conference, ACNS 2023, Kyoto, Japan, June 19–22, 2023, Proceedings, Part II. p. 516–538. Springer-Verlag, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-031-33491-7_19, https://doi.org/10.1007/978-3-031-33491-7_19
5. Bellare, M., Pointcheval, D., Rogaway, P.: Authenticated key exchange secure against dictionary attacks. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 139–155. Springer, Heidelberg (May 2000). https://doi.org/10.1007/3-540-45539-6_11
6. Bourdrez, D., Krawczyk, D.H., Lewi, K., Wood, C.A.: The OPAQUE Asymmetric PAKE Protocol. Internet-Draft draft-irtf-cfrg-opaque-10, Internet Engineering Task Force (Mar 2023), https://datatracker.ietf.org/doc/draft-irtf-cfrg-opaque/10/
7. Eaton, E., Stebila, D.: The "quantum annoying" property of password-authenticated key exchange protocols. In: Cheon, J.H., Tillich, J.P. (eds.) Post-Quantum Cryptography - 12th International Workshop, PQCrypto 2021. pp. 154–173. Springer, Heidelberg (2021). https://doi.org/10.1007/978-3-030-81293-5_9

8. Gheorghiu, V., Mosca, M.: Benchmarking the quantum cryptanalysis of symmetric, public-key and hash-based cryptographic schemes. arXiv:1902.02332 (2019)

9. Gidney, C., Ekerå, M.: How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. Quantum **5**, 433 (2021). https://doi.org/10.22331/q-2021-04-15-433, https://doi.org/10.22331/q-2021-04-15-433

10. Gu, Y., Jarecki, S., Krawczyk, H.: KHAPE: Asymmetric PAKE from key-hiding key exchange. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part IV. LNCS, vol. 12828, pp. 701–730. Springer, Heidelberg, Virtual Event (Aug 2021). https://doi.org/10.1007/978-3-030-84259-8_24

11. Hao, F., van Oorschot, P.C.: Sok: Password-authenticated key exchange – theory, practice, standardization and real-world lessons. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security. p. 697–711. ASIA CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3488932.3523256, https://doi.org/10.1145/3488932.3523256

12. Hhan, M., Yamakawa, T., Yun, A.: Quantum complexity for discrete logarithms and related problems. Cryptology ePrint Archive, Paper 2023/1054 (2023), https://eprint.iacr.org/2023/1054, https://eprint.iacr.org/2023/1054

13. Jarecki, S., Krawczyk, H., Xu, J.: OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part III. LNCS, vol. 10822, pp. 456–486. Springer, Heidelberg (Apr / May 2018). https://doi.org/10.1007/978-3-319-78372-7_15

14. Krawczyk, H.: HMQV: A high-performance secure Diffie-Hellman protocol. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 546–566. Springer, Heidelberg (Aug 2005). https://doi.org/10.1007/11535218_33

15. NIST: Nist: Selected algorithm 2022 (2022), https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022

16. Parker, E., Vermeer, M.J.D.: Estimating the energy requirements to operate a cryptanalytically relevant quantum computer. arXiv:2304.14344 (2023)

17. Roetteler, M., Naehrig, M., Svore, K.M., Lauter, K.E.: Quantum resource estimates for computing elliptic curve discrete logarithms. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, Part II. LNCS, vol. 10625, pp. 241–270. Springer, Heidelberg (Dec 2017). https://doi.org/10.1007/978-3-319-70697-9_9

18. Schmidt, J.M.: Requirements for Password-Authenticated Key Agreement (PAKE) Schemes. RFC 8125 (Apr 2017). https://doi.org/10.17487/RFC8125, https://www.rfc-editor.org/info/rfc8125

19. Taubert, T., Wood, C.A.: SPAKE2+, an Augmented PAKE. Internet-Draft draft-bar-cfrg-spake2plus-08, Internet Engineering Task Force (May 2022), https://datatracker.ietf.org/doc/draft-bar-cfrg-spake2plus/08/, work in Progress

20. Thomas, S.: Re: [Cfrg] proposed PAKE selection process. CFRG Mailing List (Jun 2019), https://mailarchive.ietf.org/arch/msg/cfrg/dtf91cmavpzT47U3AVxrVGNB5UM/#

A1 Oracles for Proof of Theorem 2

Algorithms 6 to 8 are detailed oracles for the proof of Theorem 2 in Section 3.2.

Algorithm 6 Sim. of REGISTRATION(l)

```
1: $e_l \xleftarrow{\$} \{0,1\}^{n_1}$
2: Store $e_l$
```

Algorithm 7 Sim. of PASSIVEEXEC(l)

```
1: Get stored $e_l$; Increment $ctr_l$
2: $c_{ctr,X} \xleftarrow{\$} \{0,1\}^{n_2}$
3: $y_{ctr} \leftarrow \mathbb{Z}_p$, $Y_{ctr} \leftarrow g^{y_{ctr}}$
4: return $(Y_{ctr}, e_{ctr}), (c_{ctr,X})$
```

Algorithm 8 Sim. of GETSTATIC(l)

```
Require: $l \in \mathbb{Z}$
1: Mark $l$ corrupted; $\pi_l \leftarrow [N], \mathrm{sk}_l \leftarrow \{0,1\}^\kappa$
2: for $m = 0, \ldots, ctr_l$ do
3:     for $k_{l,m}, c_{l,m} \leftarrow \mathrm{ACTIVE}_C(l, e, Y)$
4:         $a, B, \mathrm{sk} \leftarrow \mathrm{IC}_1.D(\pi_l, e)$
5:         $X \leftarrow \mathrm{IC}_2.D(\mathrm{sk}, c_{l,m})$; $x \leftarrow \mathrm{DLOG}(g, X)$
6:         $h_X = \mathcal{H}_1(l, m, X)$, $h_Y = \mathcal{H}_1(l, m, Y)$
7:         $\sigma_{l,m,C} \leftarrow (Y \circ B^{h_Y})^{x + h_X \cdot a}$
8:         $\mathcal{H}_2(l, m, X, Y, \sigma_{l,m,C}) := k_{l,m,1}$
9:     for $k_{l,2} \leftarrow \mathrm{ACTIVE}_S(l, c)$
10:        $X \leftarrow \mathrm{IC}_2.D(\mathrm{sk}_l, c)$
11:        $a, B, \mathrm{sk}_2 \leftarrow \mathrm{IC}_1.D(\pi_l, e_l)$; $b \leftarrow \mathrm{DLOG}(g, B)$
12:        $h_X = \mathcal{H}_1(m, l, X)$, $h_Y = \mathcal{H}_1(m, l, Y_{m,l})$
13:        $\sigma_{m,l,C} \leftarrow (X \circ A^{h_X})^{y_l + h_Y \cdot b}$
14:        $\mathcal{H}_2(m, l, X, Y_{m,l}, \sigma_{m,l,S}) := k_{m,l,2}$
15: return $(\pi_l, \mathrm{sk}_l)$
```

A2 Full Proof of Theorem 1

We offer the complete proof of Theorem 1 as a sequence of game hops. First, the function *CoreMap*(C, S, sid) uses the counter $\bar{l}$ mapping to the static variables indexed with l, and $ctr_{\bar{l}}$, $ctr_{C,S}$, $ctr_{C,S,\mathrm{sid}}$ corresponding to the instances using these static variables. All variables are initialized to *zero*. The *CoreMap* works as follows: if the $ctr_{C,S,\mathrm{sid}} > 0$, the respective transcript $e_{ctr_{C,S}}, Y_{ctr_{C,S}, ctr_{C,S,\mathrm{sid}}}, c_{ctr_{C,S}, ctr_{C,S,\mathrm{sid}}}$ has been generated before and is returned. Otherwise, if $ctr_{C,S} = 0$, then this is the first interaction with registrant l. The reduction sets $ctr_{C,S} \leftarrow \bar{l}$, $ctr_{\bar{l}} \leftarrow 1$, increments $\bar{l}$, corresponding to ctr_l in the KHAPE$_{\mathrm{CORE}}$, and sets $ctr_{C,S,\mathrm{sid}} \leftarrow 1$. The oracle KHAPE$_{\mathrm{CORE}}$.PASSIVEEXEC($ctr_{C,S}$) is queried; the output stored and returned. If $ctr_{C,S} > 0$, The reduction sets $ctr_{C,S,\mathrm{sid}} \leftarrow ctr_{\bar{l}}$, increments $ctr_{\bar{l}}$ and queries KHAPE$_{\mathrm{CORE}}$.PASSIVEEXEC($ctr_{C,S}$). The output is stored in $e_{ctr_{C,S}}, Y_{ctr_{C,S}, ctr_{C,S,\mathrm{sid}}}, c_{ctr_{C,S}, ctr_{C,S,\mathrm{sid}}}$ and returned.

G_0 *(Figure 1).* This is the real protocol.

G_1 *(Passive Sessions).* The game is modified by replacing the keys k_1, k_2 with random values for passively observed sessions. Particularly, on input

Execute(C, S, sid), we set $k_1 = k_2 \leftarrow \{0,1\}^\kappa$ and compute the key confirmation values τ, γ and sessions keys using the *prf*. The adversaries oracle calls to all instances l for which Execute has been called are simulated as follows: First, the simulation invokes *CoreMap*(C, S, sid) to obtain k_1, c'_X from KHAPE$_{\mathrm{CORE}}$.Active$_C(ctr_{C,S,\mathrm{sid}}, e, Y)$, is used to compute the confirmation values τ, γ. On a Corrupt(C, S) query, the extraction calls KHAPE$_{\mathrm{CORE}}$.GetStatic$(ctr_{C,S})$ returning π_l, sk_l, which programs the key k_1 returned by a KHAPE$_{\mathrm{CORE}}$.Active oracle into the *correct* position of the random oracle $\mathcal{H}_2$. The extraction receives a, B, sk from the ideal cipher on query IC$_1$.$D(\mathrm{sk}_1, e)$ as well as the discrete logarithm b from KHAPE$_{\mathrm{CORE}}$.Dlog(B). It then computes $A \leftarrow g^a$. Let $\mathcal{P}$ be a table corresponding to all N passwords. The extraction sets $\pi \leftarrow \mathcal{P}[\mathrm{sk}_1]$, *i.e.*, the sk_1'th entry of the table and returns $\pi, (e, A, b, \mathrm{sk})$, which is a perfect simulation.

For the queries $\mathcal{H}_1(\mathrm{sid}, C, S, *)$, if the entry $ctr_{C,S,\mathrm{sid}}$ is defined, the query is forwarded to the KHAPE$_{\mathrm{CORE}}$-challenger, and the result is returned. Otherwise, a random value is sampled uniformly at random from the range of $\mathcal{H}_1$, and a table is maintained for consistent responses. $\mathcal{H}_2(\mathrm{sid}, C, S, *)$ is simulated analogous to $\mathcal{H}_1$. All queries to IC$_1$ and IC$_2$ are forwarded to the KHAPE$_{\mathrm{CORE}}$-challenger. In Section 4 the divergence $q_{\mathcal{H}_2}/p$ from this simulation, *i.e.*, the random oracle and ideal cipher queries, has already been discussed.

Finally, the adversary may query a Test or Reveal query, receiving the session key from the KHAPE$_{\mathrm{CORE}}$-game, In the first case, if a Test query has been received, extraction either simulates either G_0, if the KHAPE$_{\mathrm{CORE}}$ challenge bit is *zero*, or G_1, if the KHAPE$_{\mathrm{CORE}}$ challenge bit s is *one*. When $s = 0$, the values of e, c_X as well as τ, γ are distributed as expected (*i.e.*, as in G_0), since the keys $k_1 = k_2$ are identical and thus γ can also be computed from k_1. On the other hand, if $s = 1$, the key k_1 is chosen uniformly random as expected, and thus the key confirmation values also have the expected distribution.
In the second case, if a Reveal query has been received, key k_1 returned from the simulation is real-or-random, but would be expected to always be real, resulting in a divergence. However, from an adversary detecting this change an extraction of a winning query to the KHAPE$_{\mathrm{CORE}}$ can be provided: In order to notice the change, the adversary $\mathcal{A}$ has to query the random oracle on $\mathcal{H}_2(\mathrm{sid}, C, S, X, Y, \sigma_C)$ or $\mathcal{H}_2(\mathrm{sid}, C, S, X, Y, \sigma_S)$, both of which allow to instantly win the KHAPE$_{\mathrm{CORE}}$-game. Note that the key confirmation values returned by the aPAKE impact the advantage to win the KHAPE$_{\mathrm{CORE}}$, since even a passive execution allows to verify if a derived session key is correct. Therefore, the term $\min(q_{\mathrm{AE}_C} + q_{\mathrm{AE}_S}, 1)$ is 1. Further, the inputs to *CoreMap*.Active$_*$ are sampled in KHAPE$_{\mathrm{CORE}}$.PassiveExec such that no *new* group elements, the discrete logarithm of which is knowable to the adversary, have to be considered in the probability to win the KHAPE$_{\mathrm{CORE}}$-game. Consequently, the number of these queries is exactly the number of Execute queries. The probability to detect the

difference between game G_0 and G_1 is then bounded by

$$\frac{q_{\text{DLOC}}}{N} + \epsilon_{passiv} + \frac{q_{\mathcal{H}_2}}{p} \tag{15}$$

$$\epsilon_{passiv} := \frac{(q_{\text{IC}_1} + q_{\text{IC}_2} + q_\circ)^2 + (q_{\text{DLOC}} q_\circ^2)}{p} + \frac{q_{\text{IC}_1}^2 + q_{\text{EXECUTE}}}{2^{n_1}} + \frac{q_{\text{IC}_2}^2 + q_{\text{EXECUTE}}}{2^{n_2}}\,. \tag{16}$$

G_2 (Active Sessions). In G_2, the modifications(*i.e.*, replacing k_1, k_2 with random strings), are extended to active sessions:

For an active session impersonating a client C, the oracle calls are modified as follows: On input $\text{SEND}(C, l, M = (S, \text{sid}))$ the simulation responds with the values e, Y retrieved from $\text{KHAPE}_{\text{CORE}}.\text{PASSIVEEXEC}$. On input $\text{SEND}(C, l, M = (S, \text{sid}, c_X, \tau))$ we sample the $k_2 \leftarrow \{0,1\}^\kappa$ uniformly at random and computes $\tau' \leftarrow prf(k_2, 1)$. The session key and the key confirmation value are generated from k_1, k_2 based on $\tau = \tau'$ as in an genuine execution of the protocol.

For an active session impersonating a server S, the oracle calls are modified as follows: On input $\text{SEND}(C, l, M = (S, \text{sid}, e, Y))$ the simulation samples a uniformly random value for $k_1 \leftarrow \{0,1\}^\kappa$ and computes the key confirmation value τ using the *prf*. On input $\text{SEND}(C, l, M = (S, \text{sid}, \gamma))$ we compute $\gamma' \leftarrow prf(k_1, 2)$ and set the session key conditionally on the outcome of $\gamma = \gamma'$ (*i.e.*, as in the *real* protocol). On queries to the random oracle, ideal cipher, REVEAL and CORRUPT the reduction behaves identical to G_1, and thus the divergence is identical.
Eventually, the adversary may query a TEST or REVEAL query receiving a session key from the $\text{KHAPE}_{\text{CORE}}$. To bound the adversaries chance to detect the modification, a similar extractor of a winning query to the $\text{KHAPE}_{\text{CORE}}$-game is provided. Similarly to Appendix A2, the reduction calls *CoreMap* to map instances of the QA-BPR-game to instances of the $\text{KHAPE}_{\text{CORE}}$-game. The extraction of a winning query on the adversary SEND queries to clients and servers is examined separately.

Impersonation of clients: On $\text{SEND}(C, l, M = (S, \text{sid}))$ the extraction calls $CoreMap(C, S, \text{sid})$, which causes $ctr_{C,S}$ to become defined if it previously was not, and the retrieved values e, Y are returned. On $\text{SEND}(C, i, M = (S, \text{sid}, c_X, \tau))$ the reduction calls $CoreMap(C, S, \text{sid})$ to subsequently obtain $k_2 \leftarrow \text{KHAPE}_{\text{CORE}}.\text{ACTIVE}(ctr_{C,S}, c_X)$. The key confirmation value τ' is computed from the obtained key using the *prf*. The session key and key confirmation value are set conditioned on $\tau = \tau'$ as in the real protocol.

Impersonation of Server: On $\text{SEND}(S, i, M = (C, \text{sid}, j, e, Y))$, the reduction calls $CoreMap(C, S, \text{sid})$, which causes $ctr_{C,S}$ to become defined it it previously was not. Then the reduction calls $k_1 \leftarrow \text{KHAPE}_{\text{CORE}}.\text{ACTIVE}(ctr_{C,S}, e, Y)$ and computes the key confirmation value τ genuinely using the *prf*, and returns c_X, τ. On $\text{SEND}(S, i, M = (C, j, \gamma, \text{sid}))$, the reduction computes γ' from the key k_2 using the *prf* and compares this to γ. If they match, the session key is set to $\text{K}_1 \leftarrow prf(k_1, 0)$, and otherwise, to $\perp$. For SEND the arguments are analogous to G_1: If TEST was queried, the reduction simulates G_1 (and thus G_0) perfectly if the $\text{KHAPE}_{\text{CORE}}$ challenge bit $s = 0$, and simulates G_2 if $s =$

1(except for inconsistencies in the random oracle). Otherwise, the adversary can detect the change only by querying the random oracle on either of the two inputs $\mathcal{H}_2(\mathrm{sid}, C, S, X, Y, \sigma_C)$ or $\mathcal{H}_2(\mathrm{sid}, C, S, X, Y, \sigma_S)$, both of which are winning queries for the reduction in KHAPE$_{\mathrm{CORE}}$.

The number of Active queries for which the adversary may choose the input is bounded by the number of Send queries, bounding the difference between game G_1 and G_2 by

$$\frac{(q_{\mathrm{DLOG}} + q_{\mathrm{SEND}})}{N} + \epsilon_{activ} + \frac{q_{\mathcal{H}_2}}{p} \tag{17}$$

with

$$\epsilon_{activ} := \frac{(q_{\mathrm{IC}_1} + q_{\mathrm{IC}_2} + q_{\circ})^2 + (q_{\mathrm{DLOG}} q_{\circ}^2)}{p} + \frac{q_{\mathrm{IC}_1}^2 + q_{\mathrm{SEND}}}{2^{n_1}} + \frac{q_{\mathrm{IC}_2}^2 + q_{\mathrm{SEND}}}{2^{n_2}} \tag{18}$$

.

G_3 *(Random Sessions Keys).* The final modification in G_3(*i.e.*, replacing the session keys with random strings) was discussed in Section 4, resulting in the term $(q_{\mathrm{EXEC}} + q_{\mathrm{SEND}})\epsilon_{prf}$. The sessions keys are now uniformly random and independent of the password and credentials leaving adversary to a guessing attack.

The probability that the adversary can distinguish G_0 from G_3 is bounded by

$$\frac{(q_{\mathrm{DLOG}} + q_{\mathrm{SEND}})}{N} + \frac{q_{\mathcal{H}_2}}{p} + (q_{\mathrm{EXEC}} + q_{\mathrm{SEND}})\epsilon_{prf} + \epsilon\,, \tag{19}$$

with

$$\epsilon \leq \frac{(q_{\mathrm{IC}_1} + q_{\mathrm{IC}_2} + q_{\circ})^2 + (q_{\mathrm{DLOG}} q_{\circ}^2)}{p} + \frac{(q_{\mathrm{IC}_1}^2 + q_{\mathrm{SEND}} + q_{\mathrm{EXEC}})}{2^{n_1}} + \frac{(q_{\mathrm{IC}_2}^2 + q_{\mathrm{SEND}} + q_{\mathrm{EXEC}})}{2^{n_2}}\,. \tag{20}$$

This conclude the proof. □