A compressed version of this paper appears in the proceedings of PQCrypto 2021. This is the full version.

The "quantum annoying" property of password-authenticated key exchange protocols

Edward Eaton[1,2] and Douglas Stebila[1]

[1] University of Waterloo, Waterloo, Canada,
eeaton@uwaterloo.ca, dstebila@uwaterloo.ca
[2] ISARA Corporation, Waterloo, Canada

Abstract. During the Crypto Forum Research Group (CFRG)'s standardization of password-authenticated key exchange (PAKE) protocols, a novel property emerged: a PAKE scheme is said to be "quantum-annoying" if a quantum computer *can* compromise the security of the scheme, but only by solving one discrete logarithm for each guess of a password. Considering that early quantum computers will likely take quite long to solve even a single discrete logarithm, a quantum-annoying PAKE, combined with a large password space, could delay the need for a post-quantum replacement by years, or even decades.
In this paper, we make the first steps towards formalizing the quantum-annoying property. We consider a classical adversary in an extension of the generic group model in which the adversary has access to an oracle that solves discrete logarithms. While this idealized model does not fully capture the range of operations available to an adversary with a general-purpose quantum computer, this model does allow us to quantify security in terms of the number of discrete logarithms solved. We apply this approach to the CPace protocol, a balanced PAKE advancing through the CFRG standardization process, and show that the $\text{CPace}_{\text{base}}$ variant is secure in the generic group model with a discrete logarithm oracle.

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

1 Introduction

Password-authenticated key exchange protocols, or PAKEs, are used in scenarios where public key infrastructure is unavailable, such as client-to-server authentication. Without public keys, authentication comes from a password provided by the user. This puts the security of PAKEs in an interesting place. These passwords are assumed to have low entropy, so it is possible for a malicious adversary to perform brute-force searches over the password space. The challenge in designing PAKEs is to obtain the maximum amount of security possible, despite the fact that authentication comes from low-entropy passwords. One important property of PAKEs is resistance against offline dictionary attacks: if a passive adversary observes an honest session, they still should not have enough information to break security via a brute-force search through the password space. Moreover, for an online adversary sending messages to a target session, each interaction should allow for only a single guess of the password. Thus, despite relying on low-entropy secrets, a secure PAKE can only be compromised with many online interactions, which would hopefully be noticed and stopped by a participant.

In 2019, the Crypto Forum Research Group (CFRG) issued a call for candidate password-authenticated key exchange protocols to be recommended for use in IETF protocols [14]. Both balanced PAKEs (where both parties share a password) and augmented PAKEs (where one party only has information derived from the password) were considered. Four balanced and four augmented PAKEs were considered, and in early 2020 the balanced PAKE CPace and the augmented PAKE OPAQUE were selected as recommended for usage in IETF protocols [13].

As PAKEs inherently can only be as secure the as the entropy of the password space allows, extremely detailed and fine-grained security analysis of each scheme was a focus of the selection process. In discussing potential security properties, Thomas proposed the notion of a PAKE being "quantum annoying" [15]. If a scheme is quantum annoying, then being able to solve discrete logarithms does not immediately provide the ability to compromise a system; instead, each discrete logarithm an adversary solves only allows them to eliminate a single possible password. Essentially, the adversary must guess a password, solve a discrete logarithm based on their guess, and then check to see if they were correct. This property became a topic of frequent discussion throughout the process.

CPace tries to be quantum annoying by having the base used for the Diffie–Hellman key exchange be a group element derived from the password: the parties exchange $U = g_{pw}^{u}$ and $V = g_{pw}^{v}$, and the shared secret is (roughly speaking) g_{pw}^{uv}. Seeing U and V does not yield any information about the password, since in a prime order group for every pw' there exists a u' such that $g_{pw}^{u} = g_{pw'}^{u'}$. For a quantum adversary to check a password against a transcript, it could pick a password guess pw, compute $u = \mathsf{DLOG}(g_{pw}, U)$, then check if V^u matches the session key. CPace would be quantum annoying if this is the best way to check passwords. (The other PAKE recommended by CFRG, OPAQUE, is not known to be quantum annoying.)

Current estimates for how long quantum computers will take to solve a cryptographically relevant discrete logarithm problem vary depending on factors such as the error rate and the number of coherent qubits available. In a recent analysis, Gheorghiu and Mosca [6] estimated that, to solve a discrete logarithm on the NIST P-256 elliptic curve, it would take one day on a quantum computer with 2^{26} physical qubits, or a 6 minutes on a 2^{34}-physical-qubit quantum computer. With early quantum computers taking hours or days, and even mature ones taking minutes for a single discrete logarithm, brute-forcing passwords in a quantum-annoying scheme is probably infeasible for all but the most dedicated and resourceful adversary. For well-chosen passwords from high entropy spaces, considerable quantum resources would be needed to compromise a single password. In such a scenario it would of course be best to replace PAKEs with a suitable post-quantum primitive, but quantum annoyingness is still appealing.

However, there has thus far been little formal discussion or analysis of this property. The perceived quantum annoyingness of each PAKE candidate was evaluated as part of the recommendation process, but no proof for any scheme was provided. In fact, there have been few efforts to even provide a formal definition. Quantum security models are notoriously tricky to define and use in security proofs, especially when trying to consider the cost of using Shor's algorithm [11]. Clarifying what quantum annoyingness really means and establishing how the property can be assessed for a real scheme has thus remained an open problem.

Our Contributions. In this work, we take the first steps towards putting the quantum annoying property on solid theoretical foundations. There are many difficulties in working within a fully quantum security model. Besides the typical challenges in proving security in the quantum random oracle model [4], it is not even clear what problem we could reduce to, or how that reduction would work, since we are considering an adversary that can solve discrete logarithms.

Modelling quantum-annoying via the generic group model with a discrete logarithm oracle. Instead, we consider a classical adversary in the generic group model [10, 12] who has access to a discrete logarithm oracle. This allows us to consider how 'quantum annoying' a scheme is by considering how many queries to the discrete logarithm oracle are needed in order to compromise security.

Part of the challenge in working with a discrete logarithm oracle is that the adversary can freely mix together group elements to prepare an oracle query of their choosing. For example, say we do not want the adversary to learn the discrete logarithm between group elements A and C. If the adversary queries the oracle to get the discrete logarithm between A and B, and then between B and C, they can calculate the target discrete logarithm without ever having queried it. One of the main technical difficulties we overcome in our proof is to construct a system that allows us to carefully account for exactly how much information the adversary has been able to extract from their discrete logarithm queries. We show that no matter how the adversary prepares their queries to the oracle, the information they get can be modelled as a linear system. In this view, questions about whether the adversary is 'aware' of the discrete logarithm between any two group elements can be reduced to questions on whether certain vectors appear in the rowspan of a matrix. The probability of certain events can in turn be reduced to question about the rank of this matrix. To our knowledge, this is the first time a generic group model proof has been extended with a discrete logarithm oracle, and we think that the resulting system has an interesting structure that illuminates questions about how solutions to discrete logarithms help (or don't) with the calculation of additional discrete logarithms.

Admittedly, a classical adversary in the generic group model with a discrete logarithm oracle is not a perfect model of a quantum adversary. An innovative quantum adversary could try to invent some new quantum algorithm inspired by Shor's algorithm which does not directly take discrete logarithms. Nonetheless, our approach allows for at least some formal assessment of quantum annoyingness.

Security analysis of CPace. To make use of our techniques, we focus on the protocol CPace, which was selected by the CFRG as the balanced PAKE recommendation for use in IETF protocols. We prove that $\text{CPace}_{\text{base}}$, an abstraction of the protocol that focuses on the most essential parts, is secure in a variant of the BPR model [2].

Our analysis proceeds as follows. First, we design in Section 3.1 a cryptographic problem called $\text{CPace}_{\text{core}}$ which in some sense captures the cryptographic core of $\text{CPace}_{\text{base}}$. Next, we calculate the probability that an adversary can solve in the $\text{CPace}_{\text{core}}$ problem in the generic group model with a discrete logarithm oracle; the success probability is measured in terms of the number of online interactions with a protocol participant and the number of group operations and discrete logarithms performed. An outline of the proof is provided in Section 3.2 and the full proof is given

Client C		**Server** S
Input: sid, S		Input: sid, C
$G \leftarrow H_1(\mathsf{sid} \| pw_{C,S} \| \mathrm{OC}(C,S))$		$G \leftarrow H_1(\mathsf{sid} \| pw_{C,S} \| \mathrm{OC}(C,S))$
$u \leftarrow_\$ \mathbb{Z}_p$		$v \leftarrow_\$ \mathbb{Z}_p$
$U \leftarrow G^u$	$\xrightarrow{U}$	$V \leftarrow G^v$
$K \leftarrow V^u$	$\xleftarrow{V}$	$K' \leftarrow U^v$
Abort if $K = I_{\mathcal{G}}$		Abort if $K' = I_{\mathcal{G}}$
$\mathsf{sk} \leftarrow H_2(\mathsf{sid} \| K \| \mathrm{OC}(U,V))$		$\mathsf{sk}' \leftarrow H_2(\mathsf{sid} \| K' \| \mathrm{OC}(U,V))$
Output sk		Output sk'

Fig. 1. The CPace$_{\text{base}}$ protocol.

in Section 4. In Appendix A we provide an informal discussion on the meaning of the quantum annoying property, its limitations, and some of the design decitions that may impact it. Finally, we show in Appendix B that CPace$_{\text{base}}$ is a secure PAKE in our variant of the BPR model.

As a preview of our theorem, the probability that an adversary manages to win the game is dominated by a term $(q_C + q_D)/N$ term, where q_C is the number of online interactions, q_D is the number of discrete log oracle queries, and N is the size of the password space. This lines up exactly with the intuitive guarantees we would expect a quantum annoying system to have: guess a password and try using it in an active session, or guess a password and take a discrete logarithm based on it to see if it was the password used in a passively-observed session.

2 Background

2.1 The CPace Protocol

CPace is a balanced PAKE with a simple and effective design, based on earlier protocols SPEKE [9] and PACE [3,5]. It can (optionally) be used as a subroutine for the augmented PAKE, AuCPace [8]. The fundamental structure is for the parties, sharing a password, to hash that password to a group element G and then perform a Diffie–Hellman-like key exchange with G acting as the generator. We describe it in full in Figure 1.

We will focus on CPace$_{\text{base}}$, a theoretical variant introduced by Abdalla, Haase, and Hesse [1] that distills CPace to its most essential elements. The changes between CPace$_{\text{base}}$ and the full CPace protocol are that CPace$_{\text{base}}$ uses a (multiplicatively written) group with prime order p (instead of composite order), and assumes that the random oracle H_1 maps onto the group. This variant allows us to focus on the parts of the protocol relevant to an adversary capable of solving discrete logarithms. Aspects of the security related to the process of hashing a password to a group element have been extensively covered in analysis by Abdalla et al. [1], who also give a security proof for CPace$_{\text{base}}$ in the universal composability framework. While their proof does not have any consideration of quantum annoyingness (i.e., without considering an adversary who can compute discrete logarithms), one benefit of their proof is that it does not rely on the stronger generic group model we use here.

In a CPace$_{\text{base}}$ session, the client C and server S both have a copy of the shared password $pw_{C,S}$. They receive as input a session identifier sid, and the identifier of their peer. The session identifier is assumed to come from a higher-level protocol; in some contexts, the initiator is meant to choose an sid and provide it with the first message. We will assume that the mechanism that distributes the sid to protocol participants always distributes unique values; see Appendix B.1 for details. The parties hash the session identifier, password, and a channel identifier (which is the ordered concatenation $\textsc{oc}(C, S)$ of the identities of the parties sorted by a canonical ordered) to obtain a group element G, which they then use as the base in a Diffie–Hellman key exchange.

2.2 The Generic Group Model

The generic group model [10, 12] is a cryptographic model that idealizes groups, similar to how the random oracle model idealizes hash functions. In the random oracle model, the adversary must ask the challenger to answer all hash function queries; in the generic group model; the adversary must ask the challenger to carry out all group operations using oracle queries. Group elements are represented as random strings in $\{0, 1\}^n$; these representations give the adversary no information about the structure of the group, except what they can learn by querying for it.

The generic group model was first used to provide a lower bound on the number of queries needed to solve the Diffie–Hellman problem [12] and establish bounds on reducing the discrete logarithm to the Diffie–Hellman problem [10]. As an idealization, proofs in the generic group model justify the security of these problems against an adversary who attacks in generically, regardless of the group. In the real world schemes can fall prey to better attacks, such as the number field sieve attacking the discrete logarithm problem over finite fields. However, analyzing a cryptographic scheme in the generic group model can provide some understanding of security where there otherwise may be none available.

More recently, the generic group model has been used by Yun to consider the security of the multiple discrete logarithm problem [17]. Yun showed that solving n distinct discrete logarithm problems requires at least $O(\sqrt{np})$ group operations, which matched known generic algorithms. The question of how much harder it is to solve n instances of the discrete logarithm problem on a quantum computer, which is relevant to the quantum annoying property, remains open.

3 Generic group model proof of CPace$_{\text{core}}$

We now define the CPace$_{\text{core}}$ game, and prove an upper-bound on winning this game in the generic group model. The CPace$_{\text{core}}$ game is highly customized to go hand-in-hand with the task of proving security of the CPace$_{\text{base}}$ protocol, but the basic idea of adding a discrete logarithm oracle to the generic group model as a way to capture quantum-annoyingness may have applications beyond this specific scenario.[3]

[3] We initially started out with a much simpler game in generic group model with a discrete logarithm oracle, and planned to put most of the complexity into the AKE proof. However, as developed the AKE proof, we frequently encountered steps

3.1 CPace$_{\text{core}}$ game definition

Overview. The game takes place over a collection of instances, each indexed by an integer i. For each instance i, there are N generators $g_{i,j}$. One of these generators is picked at random (represented by a target index t_i), and a Diffie–Hellman session is initiated, picking random integers $u_i, v_i \leftarrow\$ \mathbb{Z}_p$, and calculating $U_i \leftarrow g_{i,t_i}^{u_i}$, $V_i \leftarrow g_{i,t_i}^{v_i}$. All of this is set up by calls to a NewInstance oracle. We keep track of a counter variable ctr that is incremented every time NewInstance is called to keep track of the number of instances. When NewInstance is called, the adversary can optionally provide an index $\ell \leq$ ctr. This indicates that they want the new instance to be *linked* to a previous instances; linked instances use the same target index t_i. (When we interface with a PAKE adversary, this will represent sessions being instantiated with the same password.) Note that even though the index is repeated, the set of generators is distinct.

At the beginning of the game, a challenge bit s is drawn uniformly. Eventually the adversary may call a Challenge oracle with an instance i, a group element W, and a bit b indicating if they want to challenge the U half or the V half. If the challenge bit $s = 0$, then we provide $H(i, W^{u_i}, \text{OC}(U_i, W))$ or $H(i, W^{v_i}, \text{OC}(V_i, W))$ depending on which half the adversary chose to challenge. If the challenge bit $s = 1$, then the response they receive is drawn uniformly from $\mathcal{C}$ instead. The adversary is allowed to query Challenge twice per instance, once each for the U and V halves. The main challenge of the adversary is to determine the challenge bit s by trying to figure out the Diffie–Hellman completion without knowing which target index was used.

The interface with the Challenge oracle may seem somewhat arbitrary at first, with two Diffie–Hellman halves provided, and then the adversary allowed to use them separately when querying the Challenge oracle. When we interface with a real PAKE adversary in our proof of CPace$_{\text{base}}$, this simply reflects the fact that some sessions may have one or both endpoints not controlled by the adversary.

The adversary has access to a few other sources of information. The group operation $(\cdot)$ and DLOG oracles are how the adversary can find new information about group elements and the relationships between them. The GetGen oracle gives the adversary a representation of a generator for an instance and index, and the GetTarget oracle tells the adversary the target index for an instance i. In order to not make the game trivial, when GetTarget is called, we change the behaviour of the oracle H, so that whatever information the adversary was provided before is made to be consistent with H.

Details. For a positive integer m, $[m]$ represents the integers 1 through m. If $m = 0$ it represents the empty set. Define the set $\mathcal{G} \subseteq \{0,1\}^n$ to be the representation of group elements provided to an adversary, for some suitably large n. Define $\mathcal{C} = \{0,1\}^\lambda$ to be a set of confirmation values.

Parameters of the game are N, the size of the generator space; and p, the (prime) size of the group. The state of the game is maintained by a non-negative integer ctr and a bit s, with ctr initially set to 0 and s sampled uniformly from $\{0,1\}$. The adversary is given (a representation of) a generator $\mathfrak{g}$ of $\mathcal{G}$, as well as the identity element. The adversary has access to the following oracles:

where the only way we could see to proceed was to extend the generic group model game. Interestingly, the proof of the generic group model game often did not change very much as a result: the core idea of the proof—maintaining a linear system and checking for certain events based on the rank of a consistency matrix—was robust for the many features we added to the CPace$_{\text{core}}$ problem.

- $\cdot : \mathcal{G} \times \mathcal{G} \to \mathcal{G}$: The group operation oracle.
- $\mathsf{DLOG} : \mathcal{G} \times \mathcal{G} \to \mathbb{Z}_p$: A discrete logarithm oracle.
- $H : [\mathsf{ctr}] \times \mathcal{G} \times \mathcal{G} \times \mathcal{G} \to \mathcal{C}$: A confirmation value oracle. This acts as a random oracle, taking in a counter, a Diffie–Hellman completion K, and the ordered concatenation of two group elements, and returns a uniformly random group element.
- $\mathsf{GetGen} : [\mathsf{ctr}] \times [N] \to \mathcal{G}$: On input (i, j), returns $g_{i,j}$.
- $\mathsf{NewInstance} : [\mathsf{ctr}] \cup \{\perp\} \to \mathcal{G} \times \mathcal{G}$: This oracle creates a new instance of the problem. If the input is $\perp$, a new instance independent from all previous instances is generated:
  1. Increment ctr.
  2. Sample fresh generators $g_{\mathsf{ctr},j} \leftarrow\!\!\$\, \mathcal{G}$ for $j \in [N]$.
  3. Sample a uniform target index $t_{\mathsf{ctr}} \leftarrow\!\!\$\, [N]$.
  4. Sample uniform $u_{\mathsf{ctr}}, v_{\mathsf{ctr}} \leftarrow\!\!\$\, \mathbb{Z}_p$ and compute $U_{\mathsf{ctr}} \leftarrow g_{\mathsf{ctr},t_{\mathsf{ctr}}}^{u_{\mathsf{ctr}}}$, $V_{\mathsf{ctr}} \leftarrow g_{\mathsf{ctr},t_{\mathsf{ctr}}}^{v_{\mathsf{ctr}}}$.
  5. Return $U_{\mathsf{ctr}}, V_{\mathsf{ctr}}$.

  If the input is $\ell \leq \mathsf{ctr}$, the instance has the same target index as instance ℓ. The same steps are repeated but the same target index as that of session ℓ is used: step 3 is replaced by $t_{\mathsf{ctr}} \leftarrow t_\ell$. This instance is said to be *linked* to instance ℓ, as well as all other instances that instance ℓ is linked to.
- $\mathsf{Challenge} : [\mathsf{ctr}] \times \{0, 1\} \times \mathcal{G} \to \mathcal{C}$: On input $(i, b, W_{i,b})$, if $b = 0$ we calculate $K \leftarrow W_{i,0}^{u_i}$, and if $b = 1$, $K \leftarrow W_{i,1}^{v_i}$. If K is equal to the identity element, return $\perp$. Otherwise if the challenge bit $s = 0$ or $\mathsf{GetTarget}$ has been called on this or a linked instance, then return $H(i, K, \textsc{oc}(U_i, W_{i,b}))$ or $H(i, K, \textsc{oc}(V_i, W_{i,b}))$ depending on b. If $s = 1$ and $\mathsf{GetTarget}$ has not been called on a linked instance, return a randomly sampled $h_i \leftarrow\!\!\$\, \mathcal{C}$. This oracle can only be called twice per instance i, once with $b = 0$ and once with $b = 1$.
- $\mathsf{GetTarget} : [\mathsf{ctr}] \to [N]$: Returns the target index t_i for instance i. If $s = 1$, then for each instance linked to instance i, we reprogram H to behave correctly: modify H so that $H(i, W_{i,0}^{u_i}, \textsc{oc}(U_i, W_{i,0})) = h_{i,0}$, and $H(i, W_{i,1}^{v_i}, \textsc{oc}(V_i, W_{i,1})) = h_{i,1}$, where $h_{i,b}$ is the value that was previously provided for the challenge. If $\mathsf{Challenge}$ has not yet been called for one of the linked instances, then eventually is, H will skip the check for the value of s and always return the output determined by H.

The adversary wins if any of three conditions is met:

1. The adversary queries $H(i, W_{i,0}^{u_i}, \textsc{oc}(U_i, W_{i,0}))$ after making a $\mathsf{Challenge}(i, 0, W_{i,0})$ query, but before making a $\mathsf{GetTarget}$ query on a linked instance.
2. The adversary queries $H(i, W_{i,1}^{v_i}, \textsc{oc}(V_i, W_{i,1}))$ after making a $\mathsf{Challenge}(i, 1, W_{i,1})$ query, but before making a $\mathsf{GetTarget}$ query on a linked instance.
3. At the end of the game, the adversary guesses s correctly.

We want to determine the probability of the adversary's success in terms of the number of queries they make. We count the number of queries as follows:

- $q_{\mathcal{G}}$, the number of queries to the group operation oracle.
- q_D, the number of queries to the discrete logarithm oracle.
- q_N, the number of queries to $\mathsf{NewInstance}$ (i.e., the total number of instances).
- q_C, the number of queries to $\mathsf{Challenge}$ where the adversary did *not* submit $(i, 0, V_i)$ or $(i, 1, U_i)$. In other words, the number of instances for which the adversary actively participated in the Diffie–Hellman session, rather than passively observed one.
- q_G, the number of queries to GetGen.

While there are two conditions under which the adversary wins, in truth, they are one and the same. The only way to find information on the challenge bit s is to detect if the output of H is correct or not for a given instance. If a $\mathsf{GetTarget}$ query is made, then the output of H changes to no longer depend on s for that or any linked instance, and so the relevant query must be made prior to a $\mathsf{GetTarget}$ query. Thus the advantage of the adversary is entirely quantified by their ability to query $\mathsf{Challenge}(i, b, W_{i,b})$ and then either $H(i, W_{i,0}^{u_i}, \mathrm{OC}(U_i, W_{i,0}))$ or $H(i, W_{i,1}^{v_i}, \mathrm{OC}(U_i, W_{i,1}))$ before ever making a $\mathsf{GetTarget}(i)$ query.

The heart of the proof comes from the fact that even though $\mathsf{NewInstance}$ gives the adversary $U_i = g_{i,t_i}^{u_i}$ and $V_i = g_{i,t_i}^{v_i}$, it does not actually leak any information about what index t_i was used for instance i. We can write the elements of G in terms of the generator provided to the adversary, $\mathfrak{g}$. The N generators for instance i can be rewritten as

$$g_{i,1} = \mathfrak{g}^{p_{i,1}}, g_{i,2} = \mathfrak{g}^{p_{i,2}}, \ldots, g_{i,N} = \mathfrak{g}^{p_{i,N}} \ .$$

In this view, choosing a random generator corresponds to setting $U_i = \mathfrak{g}^{p_{i,t_i} \cdot u_i}$, for a random u_i and t_i. But note that *each* generator and corresponding p_{i,t_i} value is equally possible, as $p_{i,t_i} \cdot u_i = p_{i,j} \cdot (p_{i,j}^{-1} p_{i,t_i} u_i)$. Thus the only way for the adversary to proceed is to guess the generator, compute the $W_{i,0}^{u_i}$ value and query it to H. However each guess requires the adversary to know the discrete logarithm of either U_i, V_i, or $W_{i,b}$ with respect to the generator $g_{i,j}$. This requires either a discrete logarithm query to be made, or for the $W_{i,b}$ value to have been crafted so that $\mathsf{DLOG}(g_{i,t_i}, W_{i,b})$ is known to the adversary. We will therefore establish that each query to DLOG and each customised query to $\mathsf{Challenge}$ essentially provides one guess for the target index t_i, so in expectation an adversary must make roughly N such queries.

Other than this, there are small terms in the upper bound that are related to the adversary finding collisions in the generators (and thus being able to make a single DLOG query relevant to multiple instances) and the adversary calculating discrete logarithms by making group operation, rather than DLOG, queries, both of which are divided by the group order p, which is cryptographically large.

Theorem 1. *Let $\mathcal{A}$ be an adversary in the* $\mathrm{CPace}_{\mathrm{core}}$ *game. The probability that $\mathcal{A}$ wins the game is at most*

$$\frac{1}{2} + \frac{q_D + q_C}{N} + O(q_G^2/p) + O(q_D q_G^2/p) \ .$$

3.2 Proof outline

As is typical for generic group model proofs, we will maintain a table T that translates between the (additive) secret representation of elements as numbers in $\mathbb{Z}_p$ and the (multiplicative) public representation provided to the adversary, which are random unique elements of $\{0, 1\}^n$. The secret representation of the identity element is 0, and the secret representation of the generator $\mathfrak{g}$ is 1.

For an instance i and bit b, let $W_{i,b}$ be the group element that the adversary submitted to the $\mathsf{Challenge}$ oracle for the bit b, and let $w_{i,b}$ be $\mathsf{DLOG}(g_{i,t_i}, W_{i,b})$. To provide an upper bound on the adversary's ability to guess s, we need to determine their ability to query $g_{i,t_i}^{u_i w_{i,0}}$ or $g_{i,t_i}^{v_i w_{i,1}}$ to H. Except where it is relevant, for ease of notation, we will focus on the $b = 0$ case for the adversary's challenge queries, with the understanding that an implicit 'and similarly for $b = 1$' follows.

If $\mathfrak{g}^{p_{i,t_i}} = g_{i,t_i}$, then this would mean that the adversary would be unable to make a relevant query until the secret representation $p_{i,t_i} u_i w_{i,0}$ of $g_{i,t_i}^{u_i w_{i,0}}$ is added to T. However, rather than maintaining a specific $p_{i,j} \in \mathbb{Z}_p$ as the secret representation of $g_{i,j}$, we will instead maintain a variable $X_{i,j}$. For example, say the adversary queries $g_{1,1} \cdot \mathfrak{g}$ to the group operation oracle. With a specific $p_{1,1}$ in mind such that $g_{1,1} = \mathfrak{g}^{p_{1,1}}$, the secret representation of such an element would be $p_{1,1} + 1$. Instead, we write the secret representation as the linear combination $X_{1,1} + 1$, and, if we have not seen this linear combination before, choose a new public representation for it and return that to the adversary.

Similarly, the adversary may query $g_{1,1} \cdot g_{1,1}$ to the group operation oracle. We would record $2X_{1,1}$ in the table, and assuming that this term has not appeared before, give it a random unused representation. Other generators have corresponding variables $X_{i,j}$. By making group operation oracle queries combining these terms, arbitrary linear combinations of these variables can be added into the table T. This allows us to precisely quantify the information that the adversary has obtained through the discrete logarithm oracle, which in turn will allow us to precisely calculate the probability that the adversary is capable of causing certain events to happen, like making relevant queries to H.

On the other hand, the discrete logarithm oracle informs the adversary of the relationship between those linear combinations. For example, if the adversary has used the group operation oracle to figure out the representation of $\mathfrak{g}^c$ and $\mathfrak{g}^d$, and queries these to the discrete logarithm oracle, they must be provided with $c^{-1}d \bmod p$. Of course such a query provides no additional information to the adversary as they could compute it themselves. Useful queries to the discrete logarithm oracle involve group elements given to the adversary from the NewInstance or GetGen oracles. If the query $\mathsf{DLOG}(\mathfrak{g}, g_{i,j})$ is made, then a value for the corresponding $X_{i,j}$ must be decided and provided as a response.

In order to query H with the completion of the Diffie–Hellman-like session, at least one of $\mathsf{DLOG}(g_{i,t_i}, U_i)$, $\mathsf{DLOG}(g_{i,t_i}, V_i)$, $\mathsf{DLOG}(g_{i,t_i}, W_{i,b})$ must be defined. If all are undefined, the completion is undefined as well, and not possible to query. In our table T, the secret representation of U_i should be $u_i X_{i,t_i}$. However, we will instead choose a constant $\mu_i \in \mathbb{Z}_p$ and set that to be the secret representation of U_i (the secret representation of V_i will be ν_i). This means that until the adversary makes a DLOG query that causes X_{i,t_i} to become defined, u_i will not be defined.

When the adversary makes a Challenge query for an instance i, they choose a bit b and submit a group element $W_{i,b}$. In our table T, $W_{i,b}$ will have a secret representation of some linear combination of the $X_{i,j}$ variables, plus a possible constant. We must also consider the adversary's ability to cause $\mathsf{DLOG}(g_{i,t_i}, W_{i,b})$ to become defined. Essentially, the adversary will get one guess per challenge query. The adversary can select an index j and hope that $j = t_i$. Then they can construct the challenge so that they know the $w_{i,b}$ such that $W_{i,b} = g_{i,t_i}^{w_{i,b}}$, in which case the discrete logarithm is defined and the adversary can complete the challenge. We will establish however, that the adversary will only get one such guess out of the Challenge queries that they craft themselves to try and make the discrete logarithm defined.

So, the overall idea of the proof is that queries to the group operation oracle populate the table T with linear expressions and the discrete logarithm oracle enforces linear relationships between those expressions. With enough queries to the discrete logarithm oracle, the adversary can force enough relations between the various $X_{i,j}$ values that each one is entirely decided. But unless the value of X_{i,t_i} has been defined by making the proper queries to the discrete logarithm oracle, or the adversary manages to guess t_i

when making a challenge oracle query, there is no way for the adversary to query $g_{i,t_i}^{u_i w_{i,b}}$ to H, as that value is undefined, and thus has not been given a public representation.

Each query to the DLOG oracle imposes at most one linear constraint on the $X_{i,j}$ variables. Since any given j is not more likely than any other from the adversary's perspective, we need to consider the expected number of linear constraints that need to be put on the X variables before X_{i,t_i} is defined. We will show that the probability X_{i,t_i} is defined after q_D queries to the discrete logarithm oracle is at most q_D/N, which corresponds exactly to picking one session and performing a brute force search of computing the discrete logarithm of $g_{i,1}, g_{i,2} \dots$. (Viewed as a PAKE, this matches the quantum annoying property exactly: the adversary guesses the password, computes the generator that corresponds to that password, and finds the discrete logarithm with respect to that generator to make a guess towards the secret key.)

The remaining terms in the theorem's bound, $O((q_D q_G^2 + q_G^2)/p)$ come from the adversary's ability to distinguish that the oracle has been managed with unknown $X_{i,j}$ variables, rather than 'real' secret representations. The $O(q_G^2/p)$ term comes from the fact that we will provide each generator with a unique representation, while in the real world, we would expect there to eventually be collisions in the representation of the generators.

The numerator in the other term, $q_D q_G^2$, is asymptotically the same as the number of queries to the group operation oracle required to calculate a discrete logarithm (e.g., using the baby-step giant-step algorithm). So in our model, if the adversary uses the group operation oracle to calculate a discrete logarithm, rather than the provided discrete logarithm oracle, then they may notice that the discrete logarithm oracle is not behaving entirely faithfully. This happens because, when calls to the discrete logarithm oracle are made, the values of the $X_{i,j}$ can become defined. If enough group elements have been added to the table T, then it is possible that when an $X_{i,j}$ becomes defined, two of the linear polynomials in T will take on the same value in $\mathbb{Z}_p$, even though the adversary was given different representations in the generic group. However for large p, roughly $\sqrt{p}$ values need to be added to the table T in order to expect a collision to occur (the birthday paradox has come into effect).

Hence, as long as fewer than $O((q_G + q_D q_G^2)/p)$ queries to the group operation oracle happen, the discrete logarithm and group operation oracles will be managed in such a way that the adversary is unlikely to notice any difference.

4 Proof of Theorem 1

We now get into the specifics of the proof: how is the table T managed, what exactly are the linear relations imposed by the discrete logarithm oracle, and proofs of the bounds. Algorithms 1 through 6 provide a reference for how all of the algorithms are simulated.

The main technical points of the proof consist of: how group operation oracle queries add entries to T, how the NewInstance, GetGen, and Challenge oracles allow the adversary to begin interacting with the group, how discrete logarithm oracle queries are answered, and how we guarantee that responses to the oracles are consistent with each other and with past responses. With this in hand we show bound the probability that the discrete logarithm between g_{i,t_i} and U_i, V_i, or $W_{i,b}$ is defined after q_D queries to the discrete logarithm oracle and q_C modified queries to the Challenge oracle.

The group operation oracle and the table T. The table T is used to convert between the public representations provided to the adversary and the secret representation of

Algorithm 1 Simulating New Instance queries in GGM

```
Input: Integer $\ell \leq \mathsf{ctr}$ or $\perp$
1: Increment $\mathsf{ctr}$.
2: if input was integer $\ell$ then set the target index $t_{\mathsf{ctr}} \leftarrow t_\ell$. Mark instance $\mathsf{ctr}$ as
   linked to instance $\ell$, as wall as all instances $\mathsf{ctr}$ is linked to, and vice versa.
3: else Sample a uniform $t_{\mathsf{ctr}} \leftarrow_\$ [N]$.
4: Sample uniform $\mu_{\mathsf{ctr}}, \nu_{\mathsf{ctr}} \leftarrow_\$ \mathbb{Z}_p^2$, compute $U_{\mathsf{ctr}} \leftarrow \mathfrak{g}^{\mu_{\mathsf{ctr}}}$, $V_{\mathsf{ctr}} \leftarrow \mathfrak{g}^{\nu_{\mathsf{ctr}}}$.
5: Return $U_{\mathsf{ctr}}$, $V_{\mathsf{ctr}}$.
```

Algorithm 2 Simulating DLOG queries in GGM

```
Input: Query $(g_a, g_b) \in \mathcal{G} \times \mathcal{G}$, table $T$, matrix $D$ and row $\vec{r}$.
1: Use table $T$ to look up secret representations $g_a \hookrightarrow a_0 + a_1X_1 + \cdots + a_NX_N$ and
   $g_b \hookrightarrow b_0 + b_1X_1 + \cdots + b_NX_N$. If either secret representation doesn't appear in $T$,
   then return $\perp$.
2: Select a random $\vec{s}$ such that $D\vec{s} = \vec{r}$.
3: Compute $\delta = (\vec{a} \cdot \vec{s})^{-1}(\vec{b}\vec{s})$.
4: Compute the row $[\delta \cdot a_1 - b_1, \delta \cdot a_2 - b_2, \ldots, \delta \cdot a_N - b_N]$ and the value $b_0 - \delta a_0$.
5: Append the row to $D$ and the value to $\vec{r}$.
6: Return $\delta$, $D$, $\vec{r}$.
```

the element in the additive group $\mathbb{Z}_p$. To begin with, the table has just 2 elements in it: a generator $\mathfrak{g}$ and the identity element. The public representation of each of these elements is chosen at random from $\{0,1\}^n$. Note that it is common in generic group model proofs to choose n large enough so that we do not need to worry about collisions in our representations, or the adversary 'guessing' a group element that has not been added to T. Since new public representations are added to T by queries to the group operation and GetGen oracles, choosing $n >> \log_2(q_{\mathcal{G}} + q_G)$ is sufficient. It is also easy to check and see if a representation has already been used and, if so, re-sample. Since it is easy to choose a large enough n, and it impacts no other parts of the proof, we omit a term that considers the probability of picking the same representation twice.

The secret representation of $\mathfrak{g}$ is naturally 1, the identity element 0, and the secret representation of each $g_{i,j}$ from GetGen is represented by a variable $X_{i,j}$. When the group operation oracle is queried on elements g_a and g_b, the public representations are queried in the table to find the corresponding secret representation. If no such representation exists, then the query is considered invalid, and returned as such[4]. Otherwise, the secret representation of g_a and g_b will be two linear combinations of the $X_{i,j}$ variables as well as a possible constant, which we can write as $g_a \hookrightarrow a_0 + \sum_{i \in [q_N]} \sum_{j \in [N]} a_{i,j}X_{i,j} = a_0 + \vec{a} \cdot \vec{X}$ and $g_b \hookrightarrow b_0 + \sum_{i \in [q_N]} \sum_{j \in [N]} b_{i,j}X_{i,j} = b_0 + \vec{a} \cdot \vec{X}$, with $a_{i,j}, b_{i,j} \in \mathbb{Z}_p$. We can then compute the secret representation of $g_a \cdot g_b \hookrightarrow (a_0 + b_0) + \sum_{i,j}(a_{i,j} + b_{i,j})X_{i,j}$. Once the secret representation has been computed, we can check to see if this new linear combination already exists in the table. If it does, then use the already existing

[4] This is correct behaviour so long as the representation does not later become valid. Since representations are randomly chosen, the probability that this happens is negligible in n, the bit length of the representations. As discussed, we assume n is chosen to make this probability negligible.

Algorithm 3 Simulating Challenge queries in GGM

```
Input: Instance $i \in [\mathsf{ctr}]$, bit $b \in \{0,1\}$, group element $W_{i,b} \in \mathcal{G}$.
 1: if Query starting with $i$ and $b$ has been made before then return $\perp$
 2: if Instance $i$ or a linked instance has had a GetTarget query issued then
 3:     if b = 0 then Return $H(i, W_{i,b}^{u_i}, oc(U_i, W_{i,b})$
 4:     else Return $H(i, W_{i,b}^{v_i}, oc(V_i, W_{i,b})$.
 5: else Sample a uniform $h_{i,b} \leftarrow\!\!\$\, \mathcal{C}$ and return.
```

Algorithm 4 Simulating GetTarget queries in the GGM

```
Input: Instance $i \in [\mathsf{ctr}]$.
 1: if GetTarget has never been called before on $i$ or a linked instance then
 2:     for Each instance $j$ linked to instance $i$ (including $i$) do
 3:         Mark that instance has had a GetTarget query called on a linked instance
 4:         Query $\mathsf{DLOG}(g_{j,t_j}, U_j)$ and $\mathsf{DLOG}(g_{j,t_j}, V_j)$ to cause $X_{j,t_j}, v_{j,t_j}$ to become
            defined (if not already defined). Calculate $u_j \leftarrow X_{j,t_j}^{-1}\mu_j$, $v_j \leftarrow X_{j,t_j}^{-1}\nu_j$.
 5:         if Challenge$(j, 0, W_{j,0})$ has been called then
 6:             if $H(i, W_{j,0}^{u_j}, oc(U_j, W_{j,0}))$ has been called then adversary has won game
 7:             else
 8:                 Program oracle $H$ so that $H(i, W_{j,0}^{u_j}, oc(U_j, W_{j,0}))$ returns $h_{j,0}$, the
                    response to the Challenge query.
 9:         if Challenge$(j, 1, W_{j,1})$ has been called then
10:             if $H(i, W_{j,1}^{v_j}, oc(V_j, W_{j,1}))$ has been called then adversary has won game
11:             else
12:                 Program oracle $H$ so that $H(i, W_{j,1}^{v_j}, oc(V_j, W_{j,1}))$ returns $h_{j,1}$, the
                    response to the Challenge query.
13: Return $t_i$
```

representation of the group element. If not, then we can generate a new random public representation for this new linear combination and provide it to the adversary.

This simple check will only work until the adversary begins to make discrete logarithm oracle queries. As queries to the discrete logarithm oracle impose linear relationships between the $X_{i,j}$ variables, we need to check if that linear combination *modulo the relations defined* already exists in the table. We will discuss more on this point when we explain how linear relationships between the $X_{i,j}$ variables are defined by queries to the discrete logarithm oracle. As a preview, these linear relations will be encoded into a matrix D. To check and see if two secret representations $\vec{a}$ and $\vec{b}$ actually encode the same group element, we see if $\vec{a} - \vec{b}$ is linearly independent from the rows of D. If it is, then its value is not dependent on the linear relations that have been defined, and we can conclude that these represent distinct group elements.

Note as well that the group operation oracle can be extended to allow for inverses to be calculated as well. This simply means calculating $-a_0 - \sum_{i \in [q_N]} \sum_{j \in [N]} a_{i,j} X_{i,j}$ and otherwise performing the same sequence of steps.

Oracles NewInstance*,* GetGen*, and* Challenge*.* The game begins with the adversary only aware of a single generator element and the identity element. In order to begin meaningfully interacting with the game, NewInstance must be called. When this happens,

Algorithm 5 Simulating group operation oracle queries in GGM

```
Input: Query $(g_a, g_b) \in \mathcal{G} \times \mathcal{G}$, table $T$, matrix $D$ and row $\vec{r}$.
Output: Response $g_c$, updated table $T$.
1: Use table $T$ to look up secret representations $g_a \hookrightarrow a_0 + a_1X_1 + \cdots + a_NX_N$ and
   $g_b \hookrightarrow b_0 + b_1X_1 + \cdots + b_NX_N$. If either secret representation doesn't appear in $T$,
   then return $\perp$.
2: Compute $a_0 + b_0 + \sum_{i,j}(a_{i,j} + b_{i,j})X_{i,j}$. Look up this secret representation in the
   table $T$. If it exists, return the corresponding public representation (no updates to
   $T$ necessary). Otherwise, proceed.
3: for each secret representation $F(\vec{X})$ in table $T$ do
4:     Compute row $\vec{g} = [c_{1,1} - f_{1,1}, c_{1,2} - f_{1,2}, \ldots, c_{q_N,N} - f_{q_N,N}]$ and value $e = f_0 - c_0$.
5:     Check and see if $\vec{g}$ is linearly independent from the rows of $D$. If it is, then
       proceed to the next secret representation in the table.
6:     Otherwise, there exists a linear combination of rows that adds up to $\vec{g}$, i.e., a
       row vector $\vec{h}$ such that $\vec{h} \cdot D = \vec{g}$. Compute such a $\vec{h}$.
7:     If $\vec{h} \cdot \vec{r} = -e$, then return the public representation of $F(X)$. Otherwise, proceed
       to the next secret representation.
8: Sample a new public representation $g_c$ for $C(X)$. Add $C(X), g_c$ to the table $T$ and
   return $g_c$.
```

Algorithm 6 Simulating GetGen queries in GGM

```
Input: Instance $i \in [\mathsf{ctr}]$, index $j \in [N]$, table $T$
1: if $X_{i,j}$ already appears in the table $T$ then return the corresponding public
   representation $g_{i,j}$.
2: else Sample a new public representation $g_{i,j}$ and add $X_{i,j}, g_{i,j}$ to the table $T$ and
   return $g_{i,j}$.
```

we increment ctr, and if the instance is not linked to another instance, then we sample a new target index t_{ctr} from $[N]$.

Rather than earnestly generating a Diffie–Hellman-like instance from $g_{\mathsf{ctr},t_{\mathsf{ctr}}}$, we instead sample values $\mu_{\mathsf{ctr}}, \nu_{\mathsf{ctr}} \leftarrow\!\!\$\ \mathbb{Z}_p$. We will set $U_{\mathsf{ctr}} \leftarrow \mathfrak{g}^{\mu_{\mathsf{ctr}}}$, $V_{\mathsf{ctr}} \leftarrow \mathfrak{g}^{\nu_{ctr}}$. We calculate the public representation of these elements (which may be entirely new, requiring new entries into T), and return the public representation to the adversary.

We do this, rather than sending honestly generated U_i and V_i values in order to allow the discrete logarithm between g_{i,t_i} and U_i or V_i to remain undefined. Note that this does not affect the distribution of U_i or V_i. Since u_i and v_i are chosen uniformly at random, choosing the products μ_{ctr} and ν_{ctr} uniformly matches the distribution exactly. But until X_{i,t_i} becomes defined, the discrete logarithm between g_{i,t_i} and U_i and V_i is similarly undefined.

After having created an instance, the adversary can access generators through the GetGen oracle. When $\mathsf{GetGen}(i, j)$, where $i \leq \mathsf{ctr}$, is called, we sample a new public representation and add it and $X_{i,j}$ to T. We always sample unique representations, and this does create a small incongruity with the real game. In the real game, after sampling roughly $\sqrt{p}$ generators, an adversary would expect to see repetition in the public representations. But we will always provide unique representations, no matter how many times the oracle is called. This results in a $O(\mathfrak{g}^2/p)$ term in the theorem statement, representing the adversary's ability to cause a collision in the generators.

The challenge oracle is how the adversary is able to gain an advantage in winning the game. When $\mathsf{Challenge}(i, 0, W_{i,0})$ is called, we are expected to respond with either a random $h_i \leftarrow\!\!\$\, \mathcal{C}$ or $H(i, W_{i,0}^{u_i}, oc(U_i, W_{i,0}))$. We will always respond with a random h_i, so long as $\mathsf{GetTarget}(i)$ has not been called on a related instance i. This is indistinguishable as long as the adversary does not query $W_{i,0}^{u_i}$ without having previously made a $\mathsf{GetTarget}(i)$ query. If such a query is made, we consider them to have won.

The discrete logarithm oracle and the linear relationship matrix D. For queries to the discrete logarithm oracle, we need to define what linear relations are imposed, and how future oracle responses are managed for consistency. When group elements with secret representations α and $\beta \in \mathbb{Z}_p$ are queried, the response should be a value $\delta \in \mathbb{Z}_p$ such that $\alpha \cdot \delta \equiv \beta \pmod p$. So when a group element with secret representation $\alpha = a_0 + \sum a_{i,j} X_{i,j}$ and $\beta = b_0 + \sum X_{i,j}$ are queried, by returning a value $\delta \in \mathbb{Z}_p$ we are declaring that

$$\delta\big(a_0 + \sum_{i,j} a_{i,j} X_{i,j}\big) = b_0 + \sum_{i,j} b_{i,j} X_{i,j} \;\;,$$

or equivalently,

$$\sum_{i,j} (\delta a_{i,j} - b_{i,j}) X_{i,j} = b_0 - \delta \cdot a_0. \tag{1}$$

When a discrete logarithm oracle query is made, we thus need to choose a value δ consistent with all previous δ values provided. To do this we maintain a matrix D and a vector $\vec{r}$ that encodes all previous responses. That is, when a linear equation (1) is defined, we append the row

$$[\delta \cdot a_{1,1} - b_{1,1} \quad \delta \cdot a_{1,2} - b_{1,2} \quad \dots \quad \delta \cdot a_{q_N,N} - b_{q_N,N}] \tag{2}$$

to D and extend $\vec{r}$ by the entry $b_0 - \delta \cdot a_0$. Thus the set of responses provided to the adversary so far imposes the linear constraints $D\vec{X} = \vec{r}$, where

$$\vec{X} = [X_{1,1}, X_{1,2}, \dots, X_{1,N}, X_{2,1}, \dots, X_{q_N,N}]^T \;\; .$$

With this linear system in place, when a new query comes in, we can pick an arbitrary $\vec{s}$ such that $D\vec{s} = \vec{r}$, i.e., an arbitrary solution. This can be done by, for example, finding one solution and then choosing an arbitrary point in the kernel of D. Then to respond to the query $(a_0 + \vec{a} \cdot \vec{X}, b_0 + \vec{b} \cdot \vec{X}$, we can replace the $X_{i,j}$ values with the random $s_{i,j}$ values, and respond with $\delta = (a_0 + \vec{a} \cdot \vec{s})^{-1}(b_0 + \vec{b} \cdot \vec{s})$. We then add the row from (2) to D and append $b_0 - \delta \cdot a_0$ to $\vec{r}$. Our new answer is guaranteed to be consistent with all previous responses as it is consistent with $\vec{s}$, which was chosen from the solution space.

This also allows us to tell if a given $a_0 + \vec{a} \cdot \vec{X}$ has a value determined by D and $\vec{r}$, and if so, what that value is. If we can construct a linear combination of the rows of D that add up to $\vec{a}$, then the value of the linear combination is determined. Let $\vec{w}$ be the linear combination of rows, so that $\vec{w}^T D = \vec{a}^T$. Then the matrix D is telling us that $\vec{a} \cdot \vec{X} = (\vec{w}^T D)\vec{X} = \vec{w}^T(D\vec{X}) = \vec{w}^T \vec{r} = \vec{w} \cdot \vec{r}$. Thus the value (in $\mathbb{Z}_p$) of $a_0 + \vec{a} \cdot \vec{X}$ is $a_0 + \vec{w} \cdot \vec{r}$, where $\vec{w}$ is the linear combination of the rows of D that add up to $\vec{a}$. If there is no such linear combination, i.e., $\vec{a}$ is not in the rowspace of D, then the value of $a_0 + \vec{a} \cdot \vec{X}$ is not yet determined by D and $\vec{r}$. When D and $\vec{r}$ do determine a secret representations value in $\mathbb{Z}_p$, we will write it as $\cong$. So if $\vec{w}^T D = \vec{a}^T$, then $a_0 + \vec{a} \cdot \vec{X} \cong a_0 + \vec{w} \cdot \vec{r}$.

Now we may discuss how we check if a linear combination modulo the linear constraints has already appeared in the table T. When a group operation oracle query is made that will add the secret representation $a_0 + \vec{a} \cdot \vec{X}$ to the table T, we consider the difference $\vec{a} - \vec{b}$ between $\vec{a}$ and the coefficients of every other linear combination of $X_{i,j}$ values in the table T, $\vec{b}$. For each difference $\vec{a} - \vec{b}$ we need to check to see if the linear relations set forth by D mean that the new group element $a_0 + \vec{a} \cdot \vec{X}$ is actually the same as $b_0 + \vec{b} \cdot \vec{X}$.

To do this, we check to see if the rank of the matrix D is increased by appending $\vec{a} - \vec{b}$ as a row. If the rank does increase, this tells us that the relation between $\vec{a}$ and $\vec{b}$ is not defined by D. But if the rank does not increase, then the relation is defined. This means we can find the value $c \in \mathbb{Z}_p$ such that $\vec{a} - \vec{b} \cong c$.

If $c = b_0 - a_0$, then we know that these two group elements with secret representations $a_0 + \vec{a} \cdot \vec{X}$ and $b_0 + \vec{b} \cdot \vec{X}$ must be the same given the relations provided to the adversary by the discrete logarithm oracle. In this case, a new entry does not need to be added to the table T, and instead the public representation for the element already provided can be given. If $c \neq b_0 - a_0$, then the matrix D is telling us that $\vec{a}$ and $\vec{b}$ differ by a constant factor, but they are not the same element, and so the next $\vec{b}$ can be checked.

One counterintuitive aspect is that the group operation oracle is being simulated in a very expensive way. Each time a query is made, the simulator checks against each previous query made, resulting in quadratic expense. But this is not relevant to the bounds in the proof. We are not reducing CPace$_{\text{core}}$ to another problem, but providing an information-theoretic bound in terms of the number of oracle calls being made. Thus, it does not matter how efficient the simulator is, only that it counts the number of queries to the various oracle properly.

At this point, we have guarantees that (i) when a response to a discrete logarithm query is provided, it is consistent with all previous responses to the discrete logarithm oracle, and (ii) when a response to a group operation query is provided, it is consistent with all previous responses to both the group operation oracle and the discrete logarithm oracle. The remaining question is whether responses to the discrete logarithm oracle are consistent with the previous responses to the group operation oracle. In fact, they are not guaranteed to be so. Consider the case where the adversary enumerates through the entire group to get the representation of $\mathfrak{g}, \mathfrak{g}^2, \mathfrak{g}^3, \ldots, \mathfrak{g}^{p-1}$. These will all be given different public representations, but the representations will also be different from those given to all of the generators returned from GetGen. If a discrete logarithm query of the form $(\mathfrak{g}, g_{i,j})$ is made, a specific $p_{i,j} \in \mathbb{Z}_p$ will be provided. But we will have already given g^{p_i} a different representation than $g_{i,j}$, causing an inconsistency.

Since the discrete logarithm oracle responds with random answers from the solution space, these inconsistencies require the adversary to make an enormous number of group operation oracle queries to happen: it is only if $O(\sqrt{p})$ queries to the group operation oracle occur that we must worry about this inconsistency. We provide a full justification for this claim after briefly discussing the adversary's success probability.

We now return to the analysis of this game: what is the probability that the adversary succeeds after making discrete logarithm queries, and what is the difference between managing the group operation and discrete logarithm oracles in this way and a 'proper' way?

As discussed, the adversary must have done one of two things in order to possibly win. For a session i not linked to a session where a $\mathsf{GetTarget}$ query has been made, they must either know the discrete logarithm between g_{i,t_i} and either U_i, V_i, or $W_{i,b}$. We need to characterize when it is possible for an adversary to learn this based on the matrix D, and then provide an upper bound on the adversary's success probability in triggering that event.

Lemma 1. *Let $\vec{e}_{i,t_i}$ be the standard basis vector in $\mathbb{Z}^{q_N \cdot N}$ with a 1 in position (i, t_i) and 0 everywhere else. Let $\vec{w}$ be the vector representation of $W_{i,b}$, the row vector whose entries are the coefficients of the $X_{i,j}$ variables in the corresponding secret representation. Let D be the matrix of linear relations defined by the queries to the discrete logarithm oracle. Then the discrete logarithm between g_{i,t_i} and U_i, V_i, or $W_{i,b}$ is only defined if $\vec{e}_{i,t_i}$ appears in the rowspan of $\left[\frac{D}{\vec{w}}\right]$.*

Proof. Recall that the secret representation of g_{i,t_i} is X_{i,t_i}, and for U_i and V_i it is a randomly chosen pair $\mu_i, \nu_i \leftarrow\!\!\$\, \mathbb{Z}_p^2$. For the discrete logarithm between these two to be defined, the value X_{i,t_i} must be forced to have a specific value from the linear constraints of D. If it is not entirely constrained, then it can still take on any value in $\mathbb{Z}_p$. For it to take on a specific value, it must be the case that there is a linear combination of the rows of D that add up to $\vec{e}_{i,t_i}$.

Similarly, for $W_{i,b}$ we consider its vector representation $\vec{w}$. For the discrete logarithm to be defined, we must be able to rewrite this vector as a multiple of X_{i,t_i}. We can assume that X_{i,t_i} is undefined, since it being defined was already covered by the previous case. So, it must be possible, modulo the linear relations defined, to rewrite $\vec{w}$ as a multiple of $\vec{e}_{i,t_i}$. But 'zeroing out' the other entries of $\vec{v}$ like this means that $\vec{e}_{i,t_i}$ is in the rowspan of $\left[\frac{D}{\vec{w}}\right]$, as expected. □

Corollary 1. *Let W be the matrix whose rows consist of the vectorizations of each $W_{i,b}$ submitted to the* Challenge *oracle not equal to U_i or V_i. Then the instances i for which $W_{i,b}^{u_i}$ can be queried to H are restricted to those where $\vec{e}_{i,t_i}$ appears in $\left[\frac{D}{V}\right]$.*

This corollary allows us to calculate the overall probability of having the relevant discrete logarithms defined, and thus the probability of querying a Diffie-Hellman completion and winning the game. The rank of the matrix $\left[\frac{D}{W}\right]$ is at most the number of rows of D plus the number of rows of W, which is $q_D + q_C$, the number of Challenge queries where a customised $W_{i,b}$ was submitted. The rank also limits the number of basis vectors that can appear in the row span to the same number, so at most $q_D + q_C$ basis vectors can appear there.

This is how we can bound the probability that the adversary can submit the relevant group element to H. To do this, they need to have an instance i for which no GetTarget query has been made for any linked instance, and $\vec{e}_{i,t_i}$ is in the rowspan of $\left[\frac{D}{W}\right]$. Since no GetTarget query has been made for this instance, the distribution of which $\vec{e}_{i,j}$ basis vectors appear in the rowspan is independent of t_i. Thus the adversary has $q_D + q_C$ chances for a target basis vector to appear in the rowspan. So the overall probability that one appears can be upper-bounded as $(q_N + q_C)/N$.

Next we consider the question of whether the adversary can detect that we are not managing the group operation and discrete logarithm oracles perfectly. As mentioned, we do not ensure that responses to the DLOG oracle are perfectly consistent with all previous group operation oracle queries. With enough entries in T it is possible for the adversary to notice a discrepancy in how queries were handled. For example, say the adversary has queried for group elements with secret representation X_1 and d, and in the process of making discrete logarithm queries, the value of X_1 is set to be d. Since that happens after having queried X_1 and d, the two group elements will be given different public representations.

To determine the probability that any inconsistency occurs, we consider each pair of linear combinations in the table T, $(a_0 + \vec{a} \cdot \vec{X}, b_0 + \vec{b} \cdot \vec{X})$. For a given pair, we want to check to see if a new linear constraint added to D has made these two previously distinct elements take on the same value. This occurs if, before the discrete logarithm oracle query, $\vec{a}-\vec{b}$ was linearly independent from the rows of D, but after updating to D', it is now linearly *dependent*, and furthermore we have that $\vec{a}-\vec{b} \cong b_0 - a_0$.

For every pair of elements, the probability that this happens is at most $1/p$. To see this we will discuss the geometric structure of how linear constraints are added to D and what two intersecting elements means in this geometry.

Each row of D and $\vec{r}$ adds a linear constraint to the system. If the first row of D is $\vec{d}$ and the first entry of $\vec{r}$ is r then the solution space is constrained so that $\vec{d} \cdot \vec{X} = r$. We can view this as an affine hyperplane, an $(n-1)$-dimensional subspace of $\mathbb{Z}_n$. When a new row is added, this corresponds to adding another affine hyperplane. The solution space is the intersection of all hyperplanes.

The process of adding a new row to D is to select a random point in the solution space, and then construct a response to the adversary's query. The adversary's query can be seen as determining the direction of the affine hyperspace (i.e., the linear subspace that goes through zero), but the response is determined by choosing a random point in the solution space and offsetting the submitted linear subspace so that it goes through that random point, constructing an affine space.

Meanwhile, pairs in our table T collide if $\vec{a}-\vec{b}$ is linearly independent before a row is added, but linearly dependent after. The vector $\vec{a}-\vec{b}$ and value $b_0 - a_0$ also can be viewed as a hyperplane H. So the geometric interpretation of the linear relations $(D, \vec{r})$ forcing $(\vec{a}-\vec{b}) \cdot \vec{X}$ to be equal to $b_0 - a_0$ is that the solution space S is contained entirely within the hyperplane H.

This case occurs if, before a new hyperplane is added to the linear constraints, the solution space is not entirely within the hyperplane H, but, after the discrete logarithm oracle query, it is. As discussed, the process of adding a new hyperplane involves picking a random point in the solution space and making sure the new hyperplane goes through that point. For the resulting solution space to be entirely within H, it must be the case that the random point that is chosen is also within H. So the question becomes, how many points in the solution space S are also in H? Since it is not the case that S is entirely contained within H, it cannot be all of them. Since S is generated by the intersection of a series of affine hyperplanes, the intersection between that and H must be either empty, or is at most a fraction $1/p$ of the space S, as desired. This is because the intersection of such hyperplanes is an affine subspace with smaller dimension. Our base field is $\mathbb{Z}_p$, and so the subspace must have a size a power of p.

So each time a new linear constraint is added to D and $\vec{r}$, for every two entries in the table T there is at most a $1/p$ chance that these two entries now represent the same group element, modulo these constraints. Since this happens for each pair in the table, we can upper bound the overall probability of any collision happening as $O(q_{\mathcal{G}}^2/p)$, and the probability of a collision happening on any of the q_D queries to the discrete logarithm oracle as $O(q_D q_{\mathcal{G}}^2/p)$.

Thus the probability that the adversary notices the oracles misbehaving is at most $O(q_D q_{\mathcal{G}}^2/p)$, the probability that it is noticed that generators are always unique is at most $O(q_G^2/p)$, and the probability that they win assuming they do not notice misbehaviour is at most $(q_D + q_C)/N$. So the overall probability of winning is at most

$$(q_D + q_C)/N + O((q_D q_{\mathcal{G}}^2 + q_G^2)/p) \enspace .$$

Acknowledgements. E.E. was supported by a Natural Sciences and Engineering Research Council of Canada (NSERC) Alexander Graham Bell Canada Graduate Scholarship. D.S. was supported by NSERC Discovery grant RGPIN-2016-05146 and a Discovery Accelerator Supplement.

References

1. M. Abdalla, B. Haase, and J. Hesse. Security analysis of CPace. Cryptology ePrint Archive, Report 2021/114, Jan. 2021. `https://eprint.iacr.org/2021/114`.
2. M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In B. Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155. Springer, Heidelberg, May 2000. `doi:10.1007/3-540-45539-6_11`.
3. J. Bender, M. Fischlin, and D. Kügler. Security analysis of the PACE key-agreement protocol. In P. Samarati, M. Yung, F. Martinelli, and C. A. Ardagna, editors, *ISC 2009*, volume 5735 of *LNCS*, pages 33–48. Springer, Heidelberg, Sept. 2009.
4. D. Boneh, Ö. Dagdelen, M. Fischlin, A. Lehmann, C. Schaffner, and M. Zhandry. Random oracles in a quantum world. In D. H. Lee and X. Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 41–69. Springer, Heidelberg, Dec. 2011. `doi:10.1007/978-3-642-25385-0_3`.
5. Federal Office for Information Security (BSI). Advanced security mechanism for machine readable travel documents (extended access control (EAC), password authenticated connection establishment (PACE), and restricted identification (RI)), 2008. SI-TR-03110, Version 2.0, `https://www.bsi.bund.de/EN/Service-Navi/Publications/TechnicalGuidelines/TR03110/BSITR03110.html`.
6. V. Gheorghiu and M. Mosca. Benchmarking the quantum cryptanalysis of symmetric, public-key and hash-based cryptographic schemes, 2019. `arXiv:1902.02332`.
7. B. Haase. CPace, a balanced composable PAKE. Internet-Draft draft-haase-cpace-01, Feb. 2020. `http://www.ietf.org/internet-drafts/draft-haase-cpace-01.txt`.
8. B. Haase and B. Labrique. AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT. *IACR TCHES*, 2019(2):1–48, 2019. `https://tches.iacr.org/index.php/TCHES/article/view/7384`. `doi:10.13154/tches.v2019.i2.1-48`.
9. D. P. Jablon. Strong password-only authenticated key exchange. *Comput. Commun. Rev.*, 26(5):5–26, 1996. `doi:10.1145/242896.242897`.
10. U. M. Maurer and S. Wolf. Lower bounds on generic algorithms in groups. In K. Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 72–84. Springer, Heidelberg, May / June 1998. `doi:10.1007/BFb0054118`.
11. P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. `doi:10.1137/S0097539795293172`.
12. V. Shoup. Lower bounds for discrete logarithms and related problems. In W. Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997. `doi:10.1007/3-540-69053-0_18`.
13. S. Smyshlyaev, N. Sullivan, and A. Melnikov. [Cfrg] Results of the PAKE selection process. CFRG Mailing List, Mar. 2020. `https://mailarchive.ietf.org/arch/msg/cfrg/LKbwodpa5yXo6VuNDU66vt_Aca8/`.
14. N. Sullivan, S. Smyshlyaev, K. Paterson, and A. Melnikov. Proposed PAKE selection process. CFRG Mailing List, May 2019. `https://mailarchive.ietf.org/arch/msg/cfrg/-J43ZsPw2J5MBC-k8y6--kJJtZk/`.
15. S. Thomas. Re: [Cfrg] proposed PAKE selection process. CFRG Mailing list, June 2019. `https://mailarchive.ietf.org/arch/msg/cfrg/dtf91cmavpzT47U3AVxrVGNB5UM/`.
16. L. Vaillant-David. Re: [Cfrg] CPACE: what the "session id" is for? CFRG Mailing List, June 2020. `https://mailarchive.ietf.org/arch/msg/cfrg/1b3H_VKyZR-UbE6FoNeOdOOPqRE/`.

17. A. Yun. Generic hardness of the multiple discrete logarithm problem. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 817–836. Springer, Heidelberg, Apr. 2015. doi:10.1007/978-3-662-46803-6_27.

A Limitations to Quantum Annoyingness

A.1 Forward Secrecy

Providing a discrete logarithm oracle to an adversary makes them incredibly strong, and it is impressive that some PAKEs can still achieve some manner of security in such a model. But it is important to note that not all security properties we expect a PAKE to have hold against an adversary capable of solving discrete logarithms. One such notion is that of forward secrecy. In most PAKE security models, we would expect that if two honest parties engage in a properly executed session, and then at some point later, the adversary compromises the password used, this should not impact the security of previous sessions.

This is does not necessarily hold when the adversary has access to a discrete logarithm oracle. For example in CPace, if an adversary holds the transcript $U = g_{pw}^u, V = g_{pw}^v$ for a previous session and later compromises the password pw used for that session, then they can easily calculate the generator g_{pw} used, make a single discrete logarithm calculation to compute say u, and then easily recover the session key.

This limits the types of statements we can make about quantum annoyingness. When analyzing PAKE security, we need to change the definition of a fresh session that the adversary can try to defeat. In the BPR model [2] as described in Appendix B.1, we restrict the adversary from trying to win on sessions where they have *ever* corrupted the user. Contrast this with the traditionally desired property of forward secrecy in the BPR model, where the adversary is restricted from targeting sessions where they had corrupted the user's password before the session started and then actively participated in the session, but is allowed to target sessions where a user's password is corrupted after the session completed.

Of course, forward secrecy is a desirable property. CPace has fortunately been shown to have forward secrecy against a classical adversary [1]. The attack above shows it does not have forward secrecy against an attacker with discrete logarithm powers, but our proof shows that it is at least non-forward-secret secure against such attackers (in the generic group model).

A.2 Session and Channel Identifiers

While in this text, we restrict ourselves to the specific $\text{CPace}_{\text{base}}$ protocol, CPace has a more general design with various options for how different parts of the scheme can be configured. In particular, the current CPace specification [7] is intentionally flexible about where the session ID comes from and what information is included in the channel identifier. While this is needed in order to allow the protocol to be used in more situations, it means that not all instantiations of CPace may provide the same level of quantum annoyingness.

It has been noted by participants on the CFRG mailing list that the uniqueness of the sid affects the quantum annoying property in CPace [16]. Consider a situation where both the session ID and the channel identifier are not used. For a set of N passwords $\{pw_j\}_{j\in[N]}$, all sessions and users will share the same set of generators determined by $H_1(pw_1), H_1(pw_2)$, etc. An adversary can then calculate the discrete logarithm of each generator with respect to a global generator $\mathfrak{g}$, obtaining the $p_1, p_2, \ldots$ such that $H_1(pw_1) = \mathfrak{g}^{p_1}, H_1(pw_2) = \mathfrak{g}^{p_2}, \ldots$. This allows the adversary to perform an offline dictionary attack on each user with a single Send query as follows. The adversary begins a session with a target, and receives a group element U; they respond with a group element g^x, for a random $x \leftarrow\!\!\$\ \mathbb{Z}_p$; after receiving a message encrypted under the session key, they can check if the session key equals $\mathfrak{g}^{x/p_i}$ for one of the p_i values, enabling password recovery.

This is prevented by having unique session identifiers for each session. In this case, the set of candidate generators for each session is unique, so the discrete logarithm computations do not carry over from one session to another. There remains an interesting question on what happens if the session identifier is not necessarily unique, but the channel identifier is. If this happens, then for any given pair of users, the channel identifier, and thus the set of candidate generators, is unique. But this pair of users always uses the same password in all of their sessions. Thus, an adversary's precomputation advantage would be restricted to one pair of users. In our analysis, however, we consider the case where the session identifiers are all unique.

B PAKE security of CPace$_{\text{base}}$

In this section we show that CPace$_{\text{base}}$ is a secure password-authenticated key exchange protocol in a variant of the Bellare–Pointcheval–Rogaway (BPR) model [2], assuming the difficulty of the CPace$_{\text{core}}$ problem from Section 3. Our BPR$'$ security model differs from the BPR model in that it does not provide forward secrecy, assumes a balanced PAKE (i.e., the server stores the client's password directly, not a transformation thereof), and accommodates externally specified session identifiers, in addition to providing generic group model oracles.

B.1 The BPR$'$ Model

Participants and passwords. Fix a non-empty finite set $\mathcal{C} \cup \mathcal{S}$ of participants; each participant is either a client or server, but never both. For each client-server pair (C, S), a password $pw_{C,S}$ is chosen uniformly at random from a set $\mathcal{P}$; each client and server has a copy of the passwords relevant to them.[5]

Sessions and state. Each participant P can execute multiple instances of the protocol simultaneously, each of which is called a session; sessions within a party are numbered sequentially, and the ith session at participant P is denoted π_P^i. For each session i, participant P maintains the following state variables:

- $\mathsf{acc}_P^i \in \{\mathsf{true}, \mathsf{false}\}$: whether the instance has successfully accepted a session key
- $\mathsf{term}_P^i \in \{\mathsf{true}, \mathsf{false}\}$: whether the instance has terminated, meaning no more incoming or outgoing messages
- state_P^i: private state of the protocol execution
- sid_P^i: the session identifier
- pid_P^i: the partner identifier (who U believes they are communicating with)
- sk_P^i: the session key

[5] When CPace is run inside of AuCPace [8], the CPace password is output from an earlier phase of AuCPace.

Adversarial interaction. The adversary in the security model has full control over the network. The adversary initiates all actions, controls delivery of all protocol messages, and can create, modify, delay, repeat, or delete messages. The adversary interacts with honest participants via calls to the following oracles. In square brackets at the end of each query's description is the symbol we use to denote the number of queries made to that oracle.

– $\mathsf{Send}(P, i, M)$: Captures an active attack. An adversary-selected message M is sent to instance π_P^i, which processes it based on its current state and returns any response message to the adversary. The first call to Send for each instance may include additional context information in M, such as the identity of the intended peer. [Number of Send queries: q_S]
– $\mathsf{Execute}(C, i, S, j)$: Captures the adversary's ability to passively observe honest sessions. If both π_C^i and π_S^j have not yet been used, this query executes the protocol between those two instances. The adversary is provided a transcript of the messages sent by each party. [q_E]
– $\mathsf{Reveal}(P, i)$: The session key sk_P^i is revealed, if it has been set. [q_R]
– $\mathsf{Corrupt}(C, S)$: Reveals $pw_{C,S}$ to the adversary. [q_{Co}]
– $\mathsf{Test}(P, i)$: Issues the challenge for the adversary. Flips a bit b. If $b = 1$ the session key sk_P^i is revealed to the adversary. If $b = 0$ a uniformly random session key is drawn and returned. This query can be called only once.
– $\cdot(A, B)$: The group operation oracle. [q_G]
– $\mathsf{DLOG}(A, B)$: The discrete logarithm oracle. [q_D]

The adversary's goal in the security experiment is, for a sufficiently uncompromised target sessions, to distinguish the real session key from a random one. At the end of the experiment, the adversary outputs a bit, which is its guess as to whether it was given the real session key or a random one.

Partnering and freshness. Since the adversary can compromise some values and impersonate users, we have to restrict which sessions count as a win for the adversary.

Let π_C^i be a client instance and π_S^j be a server instance with $\mathsf{acc}_C^i = \mathsf{acc}_S^j = \mathsf{true}$. We say that π_C^i and π_S^j are *partnered* if $\mathsf{pid}_C^i = S$, $\mathsf{pid}_S^j = C$, $\mathsf{sk}_C^i = \mathsf{sk}_S^j$, $\mathsf{sid}_C^i = \mathsf{sid}_S^j$, and there is no other accepting instance with the same sid.

Further, an instance π_P^i is considered *fresh* if all of the following conditions are satisfied:

– a $\mathsf{Reveal}(P, i)$ query has not been made;
– if a partnered instance $\pi_{P'}^j$ exists, then a $\mathsf{Reveal}(P', j)$ query has not been made;
– no $\mathsf{Corrupt}(C, S)$ query has occurred, where C and S are the client and server among P and pid_P^i.

Since we are aiming for security in the quantum-annoying model, we cannot hope to achieve forward secrecy as noted in Appendix A.1: in CPace$_{\text{base}}$, an adversary with a discrete logarithm oracle could $\mathsf{Execute}$ a session, Test its session key, then $\mathsf{Corrupt}$ the password, hash it to get the corresponding generator, then use its discrete logarithm oracle to find one party's ephemeral shared secret and compute the session key. Thus, our freshness condition above (specifically the third bullet point) does not capture forward secrecy.

Advantage. For a PAKE protocol Π, we say that the adversary *succeeds* if they make a single Test query to an instance that has accepted and terminated and remains fresh throughout the game, and the adversary returns a bit b' that is equal to the bit b that was sampled in the process of answering the Test query. The advantage of the adversary $\mathcal{A}$ is defined as

$$\mathrm{Adv}_{\Pi}^{\mathrm{BPR}'}(\mathcal{A}) = 2\Pr[\mathcal{A} \text{ succeeds}] - 1 \ .$$

B.2 Security of CPace$_{\mathrm{base}}$

The CPace$_{\mathrm{core}}$ problem defined in Section 3 is somewhat unnatural and rather complex, but the benefit of that complexity is that it captures in a single problem all of the characteristics needed to prove the security of CPace$_{\mathrm{base}}$ in the BPR$'$ model.

In CPace$_{\mathrm{base}}$, session identifiers are externally provided. For example, when CPace$_{\mathrm{base}}$ is run as a sub-protocol of AuCPace [8], an earlier stage of AuCPace establishes the a session identifier. For the purposes of the proof, we will assume that session identifiers are provided by the adversary to sessions, with the constraint that, for any session identifier, the adversary may initiate at most one honest client session and at most one honest server session with that session identifier. (This corresponds to the idea that each honest party contributes something fresh and unique-within-that-party to the identifier of each session they participate in, which is the case with how session identifiers are established in AuCPace.)

Theorem 2. *Let $\mathcal{G}$ be a cyclic group of prime order p, and let $H_1 : \{0,1\}^* \to \mathcal{G}$ and $H_2 : \{0,1\}^* \to \{0,1\}^\lambda$ be random oracles. Let $\mathcal{P}$ be a password space of size N. If the* CPace$_{\mathrm{core}}$ *problem is hard in the generic group model (with a discrete logarithm oracle) for $\mathcal{G}$, then the* CPace$_{\mathrm{base}}$ *protocol is secure. In particular, if $\mathcal{A}$ is an adversary against* CPace$_{\mathrm{base}}$ *in the* BPR$'$ *model, then there exists an adversary $\mathcal{B}$ against* CPace$_{\mathrm{core}}$ *such that*

$$\mathrm{Adv}_{\mathrm{CPace}_{\mathrm{base}}}^{\mathrm{BPR}'}(\mathcal{A}) \leq \frac{4q_{H_2}}{p} + \mathrm{Adv}^{\mathrm{CPace}_{\mathrm{core}}}(\mathcal{B}) \ ,$$

where $\mathcal{A}$ makes at most q_{H_2} queries to H_2. Moreover, the running time of $\mathcal{B}$ is the about the same as that of $\mathcal{A}$, and the number of queries $\mathcal{B}$ makes to its CPace$_{\mathrm{core}}$ *oracles, in terms of the number of queries $\mathcal{A}$ makes to its* CPace$_{\mathrm{base}}$ *oracles, is as follows:*

- $\cdot$ *(the group operation oracle):* $q_{\mathcal{G}}^{\mathcal{B}} = q_{\mathcal{G}}^{\mathcal{A}}$
- DLOG*:* $q_D^{\mathcal{B}} = q_D^{\mathcal{A}}$
- *H:* $q_H^{\mathcal{B}} \leq q_{H_2}^{\mathcal{A}}$
- GetGen*:* $q_G^{\mathcal{B}} \leq q_{H_1}^{\mathcal{A}}$
- NewInstance*:* $q_N^{\mathcal{B}} \leq q_E^{\mathcal{A}} + q_S^{\mathcal{A}} + q_{Co}^{\mathcal{A}} + q_{H_1}^{\mathcal{A}}$
- Challenge *queries of type 1:*[6] $q_{C1}^{\mathcal{B}} \leq q_E^{\mathcal{A}}$
- Challenge *queries of type 2:* $q_{C2}^{\mathcal{B}} \leq q_S^{\mathcal{A}}$
- GetTarget*:* $q_T^{\mathcal{B}} \leq q_{Co}^{\mathcal{A}}$.

Combining Theorem 2 with Theorem 1 yields:

[6] We distinguish Challenge queries that do submit either $(i, 0, V_i)$ or $(i, 1, U_i)$ and Challenge queries that do not submit either of those as type 1 and type 2, respectively. This is because the bounds in Theorem 1 about CPace$_{\mathrm{core}}$ only care about type 2 Challenge queries.

Corollary 2. *In the generic group model (with a discrete logarithm oracle) for a group $\mathcal{G}$ of order p, for any adversary $\mathcal{A}$ making q_{H_1} H_1 and q_{H_2} H_2 random oracle queries, q_S Send queries, $q_{\mathcal{G}}$ group operation queries, and q_D discrete logarithm queries, the advantage of $\mathcal{A}$ in breaking the security of* $\mathrm{CPace_{base}}$ *with a password dictionary of size N is at most*

$$\mathrm{Adv}^{\mathrm{BPR}'}_{\mathrm{CPace_{base}}}(\mathcal{A}) \leq \frac{q_D + q_S}{N} + \frac{4q_{H_2} + O(q_{H_1}^2 + q_D q_{\mathcal{G}}^2)}{p} \enspace .$$

Proof (of Theorem 2). We give a reduction $\mathcal{B}$ that, using a $\mathrm{CPace_{core}}$ challenger, simulates the BPR' security experiment for $\mathrm{CPace_{base}}$ to $\mathcal{A}$.

The idea behind the simulation $\mathcal{B}$ is as follows. $\mathcal{B}$ maintains a mapping ctr of how $\mathrm{CPace_{base}}$ user pairs (C, S) and matching sessions (C, S, sid) map on to $\mathrm{CPace_{core}}$ instances. Recall that calling $\mathrm{CPace_{core}}.\mathsf{NewInstance}$ with a previously used counter will cause the $\mathrm{CPace_{core}}$ instance to re-use the same target index; this will correspond to sessions between the same pair of users using the same password; and calling $\mathrm{CPace_{core}}.\mathsf{GetTarget}$ will allow $\mathcal{B}$ to answer password Corrupt queries. $\mathcal{B}$ will use the $\mathrm{CPace_{core}}.\mathsf{NewInstance}$ oracle to simulate message generation and the $\mathrm{CPace_{core}}.\mathsf{Challenge}$ oracle to compute session keys. One significant difference in $\mathcal{B}$'s simulation is that all session keys – even those returned by Reveal, not just the one returned by Test – are either real or random depending on the hidden secret s of the $\mathrm{CPace_{core}}$ game. But this will not be a problem, as detecting this in the random oracle model requires a query to the random oracle H_2 which is forwarded to the $\mathrm{CPace_{core}}.H$ oracle, and would lead to a win in the $\mathrm{CPace_{core}}$ game.

Initialize $\mathsf{ctr}^* \leftarrow 0$. Define the following subroutine:

- $\textsc{getUV}(C, S, \mathsf{sid})$:
  1. If $\mathsf{ctr}_{C,S,\mathsf{sid}}$ is defined: return $(U_{C,S,\mathsf{sid}}, V_{C,S,\mathsf{sid}})$.
  2. Else if $\mathsf{ctr}_{C,S}$ is defined: set $(U_{C,S,\mathsf{sid}}, V_{C,S,\mathsf{sid}}) \leftarrow \mathrm{CPace_{core}}.\mathsf{NewInstance}(\mathsf{ctr}_{C,S})$, increment ctr^*, and set $\mathsf{ctr}_{C,S,\mathsf{sid}} \leftarrow \mathsf{ctr}^*$. Return $(U_{C,S,\mathsf{sid}}, V_{C,S,\mathsf{sid}})$.
  3. Else: Set $(U_{C,S,\mathsf{sid}}, V_{C,S,\mathsf{sid}}) \leftarrow \mathrm{CPace_{core}}.\mathsf{NewInstance}(\perp)$, increment ctr^*, and set $\mathsf{ctr}_{C,S}$ and $\mathsf{ctr}_{C,S,\mathsf{sid}}$ to ctr^*.

$\mathcal{B}$ answers queries from $\mathcal{A}$ as follows:

- $\mathsf{Execute}(C, i, S, j, \mathsf{sid})$: We use the NewInstance (via $\textsc{getUV}$) and Challenge oracles of the $\mathrm{CPace_{core}}$ challenger to generate a transcript and session key, and receive a session identifier from the adversary, which must be previously unused by C and S. Set $\mathsf{sid}^i_C, \mathsf{sid}^j_S \leftarrow \mathsf{sid}$. Set $(U, V) \leftarrow \textsc{getUV}(C, S, \mathsf{sid})$. Set $\mathsf{sk}^i_C \leftarrow \mathrm{CPace_{core}}.\mathsf{Challenge}(\mathsf{ctr}_{C,S,\mathsf{sid}}, 0, V)$ and $\mathsf{sk}^j_S \leftarrow \mathsf{sk}^i_C$. Return transcript (U, V).
- $\mathsf{Send}(C, i, M = (\mathsf{sid}, S))$ to a client C: We use the NewInstance oracle of the $\mathrm{CPace_{core}}$ challenger (via $\textsc{getUV}$) to generate the message for the client side of a session. Set $\mathsf{sid}^i_C \leftarrow \mathsf{sid}$. Run $\textsc{getUV}(C, S, \mathsf{sid})$. Return outgoing message U^i_C.
- $\mathsf{Send}(S, j, M = (C, \mathsf{sid}, U))$ to a server S: We use the NewInstance oracle of the $\mathrm{CPace_{core}}$ challenger (via $\textsc{getUV}$) to generate the message for the server side of a session, and the Challenge oracle to generate the session key. Set $\mathsf{sid}^j_S \leftarrow \mathsf{sid}$. Run $\textsc{getUV}(C, S, \mathsf{sid})$ and set $\mathsf{sk}^j_S \leftarrow \mathrm{CPace_{core}}.\mathsf{Challenge}(\mathsf{ctr}_{C,S,\mathsf{sid}}, 1, U)$. Return outgoing message V^j_S.
- $\mathsf{Send}(C, i, M = V)$ to a client C: We use the Challenge oracle of the $\mathrm{CPace_{core}}$ challenger to complete the session. Set $\mathsf{sk}^i_C \leftarrow \mathrm{CPace_{core}}.\mathsf{Challenge}(\mathsf{ctr}_{C,S,\mathsf{sid}^i_C}, 0, V)$.

- $\mathsf{Reveal}(P, i)$: Return sk_P^i, if it has been set.
- $\mathsf{Corrupt}(C, S)$: We use the $\mathsf{GetTarget}$ oracle of the $\text{CPace}_{\text{core}}$ challenger to let the $\text{CPace}_{\text{core}}$ challenger pick which password is being used for this client-server pair. If $pw_{C,S}$ is set, return it. If $\mathsf{ctr}_{C,S}$ is not defined, run $\text{GETUV}(C, S, \mathsf{sid})$ for a random, unused sid. Let $t \leftarrow \text{CPace}_{\text{core}}.\mathsf{GetTarget}(\mathsf{ctr}_{C,S})$. Set $pw_{C,S} \leftarrow \mathcal{P}[t]$ (i.e., the tth password in password dictionary $\mathcal{P}$). Return $pw_{C,S}$.
- $\mathsf{Test}(P, i)$: Return sk_P^i.
- $\cdot(A, B)$ (the group operation oracle): Return $\text{CPace}_{\text{core}}. \cdot (A, B)$.
- $\mathsf{DLOG}(A, B)$: Return $\text{CPace}_{\text{core}}.\mathsf{DLOG}(A, B)$.
- $H_1(\mathsf{sid}\|pw\|\text{OC}(C, S))$: If $\mathsf{ctr}_{C,S,\mathsf{sid}}$ is not defined, then run $\text{GETUV}(C, S, \mathsf{sid})$. Let t be the index of pw in the password dictionary $\mathcal{P}$. Return $\text{CPace}_{\text{core}}.\mathsf{GetGen}(\mathsf{ctr}_{C,S,\mathsf{sid}}, t)$.
- $H_2(\mathsf{sid}\|K\|\text{OC}(U, V))$: If this query has already been asked, answer as before. Otherwise:
  - If there is no C, S such that $\mathsf{ctr}_{C,S,\mathsf{sid}}$ is defined, we maintain the random oracle ourselves using a table $\mathcal{H}$. If $\mathcal{H}[\mathsf{sid}\|K\|\text{OC}(U, V)]$ is not defined, select it uniformly at random from the set $\text{CPace}_{\text{core}}.\mathcal{C}$. Return $\mathcal{H}[\mathsf{sid}\|K\|\text{OC}(U, V)]$.
  - If there exists C, S such that $\mathsf{ctr}_{C,S,\mathsf{sid}}$ is defined: Return $\text{CPace}_{\text{core}}.H(\mathsf{ctr}_{C,S,\mathsf{sid}}, K, \text{OC}(U, V))$.

$\mathcal{B}$ runs $\mathcal{A}$ until either $\mathcal{B}$ wins the $\text{CPace}_{\text{core}}$ game, or $\mathcal{A}$ terminates and outputs a bit, in which case $\mathcal{B}$ uses that bit as its guess of s in the $\text{CPace}_{\text{core}}$ game.

In most ways, $\mathcal{B}$ correctly simulates the execution of $\text{CPace}_{\text{base}}$ to $\mathcal{A}$. The password $pw_{C,S}$ for a client-server pair corresponds to the tth password in the dictionary, where t is the target index of all sessions between C and S (since they use the same $\mathsf{ctr} = \mathsf{ctr}_{C,S}$ in calls to $\text{CPace}_{\text{core}}.\mathsf{NewInstance}$). The messages in the $\mathsf{Execute}$ and Send queries are distributed exactly as in $\text{CPace}_{\text{base}}$. The responses to $\mathsf{Corrupt}$, $\cdot$, DLOG, and H_1 are also all distributed correctly.

Assuming the Test session $\pi_{P^*}^{i^*}$ remains fresh means that the adversary has not made a $\mathsf{Corrupt}(C^*, S^*)$ query for the client C^* and server S^* in the test session prior to the Test query. Therefore, $\mathcal{B}$ has not made a $\mathsf{GetTarget}(\mathsf{ctr}_{C^*,S^*})$ query to $\text{CPace}_{\text{core}}$ prior to the $\mathsf{Challenge}$ query being issued for the instance corresponding to the Test session, and thus the session key in the test session is real-or-random, depending on the secret bit s of the $\text{CPace}_{\text{core}}$ game. Thus the response to the Test query is properly distributed.

The session keys set during $\mathsf{Execute}$ and Send queries are *not* perfectly simulated; in the BPR' experiment for $\text{CPace}_{\text{base}}$, only the response to the Test query should be real-or-random, but in $\mathcal{B}$'s simulation, all session keys are real-or-random (since they all are generated by a call to $\text{CPace}_{\text{core}}.\mathsf{Challenge}$) and thus all responses from Reveal are real-or-random. Additionally, responses to H_2 are not simulated correctly when called with a sid which has not been already passed to a client or server instance via $\mathsf{Execute}$ or Send. The rest of the proof focuses on why these inconsistencies are not a problem.

First we consider whether an adversary can detect that responses from a $\mathsf{Reveal}(P, i)$ query are real-or-random, not real. Since session keys are the output of the random oracle H_2, this would mean that the adversary has to query H_2 on $\mathsf{sid}_P^i\|K\|\text{OC}(U, V)$ where $K = DH(U, V)$ and U, V are the messages used by instance π_P^i; call this E_1. Let C and S be the client and server instance respectively among P and pid_P^i. If the adversary has called $\mathsf{Corrupt}(C, S)$ or $\mathsf{Corrupt}(S, C)$ prior to this Reveal query, then $\mathcal{B}$ will have called $\mathsf{GetTarget}(\mathsf{ctr}_{C,S})$, and $\text{CPace}_{\text{core}}$ is defined such that a subsequent call to $\mathsf{Challenge}$ for any instance linked to $\mathsf{ctr}_{C,S}$ returns real session keys, regardless of the hidden bit. If the adversary has not called $\mathsf{Corrupt}(C, S)$ or $\mathsf{Corrupt}(S, C)$ prior to this reveal query, then $\mathcal{B}$ will not have called $\mathsf{GetTarget}(\mathsf{ctr}_{C,S})$. Since π_P^i is a session at a party simulated by $\mathcal{B}$, either $U = U_{C,S,\mathsf{sid}_P^i}$ (if $P = C$) or $V = V_{C,S,\mathsf{sid}_P^i}$ (if $P = S$). Either way, at least one of U, V was the output of a call to $\mathsf{NewInstance}$ for $\mathsf{ctr}_{C,S,\mathsf{sid}_P^i}$. By construction, $\mathcal{B}$ relays an H_2 query for a defined sid_P^i to $\mathrm{CPace}_{\mathrm{core}}.H$. Hence, $\mathcal{B}$ queries $\mathrm{CPace}_{\mathrm{core}}.H$ with either $(i^*, W_{i^*,0}^{u_{i^*}}, \mathrm{OC}(U_{i^*}, W_{i^*,0}))$ (in the case where $P = C$, taking $i^* = \mathsf{ctr}_{C,S,\mathsf{sid}_P^i}$, $U_{i^*} = U = U_{C,S,\mathsf{sid}_P^i}$, $W_{i^*,0} = V$) or $(i^*, W_{i^*,1}^{v_{i^*}}, \mathrm{OC}(V_{i^*}, W_{i^*,1}))$ (in the case where $P = S$, taking $i^* = \mathsf{ctr}_{C,S,\mathsf{sid}_P^i}$, $V_{i^*} = V = V_{C,S,\mathsf{sid}_P^i}$, $W_{i^*,1} = U$), both of which are immediately winning queries to $\mathrm{CPace}_{\mathrm{core}}$. In other words, $\Pr[\mathsf{E}_1] \leq \epsilon$, where ϵ is the probability of winning $\mathrm{CPace}_{\mathrm{core}}$ with the number of queries made by $\mathcal{B}$.

Now we consider whether an adversary can detect that responses to H_2 are not simulated correctly when called with a sid which has not been already passed to a client or server instance via $\mathsf{Execute}$ or Send; call this E_2. We permitted the adversary to choose sids for sessions, so we will not assume that sids are unpredictable before used in an session at an honest party. However, when a session is activated at an honest party using either $\mathsf{Execute}$ or Send, a fresh U or V value (depending on whether it is a client or server session) will be chosen by the simulator. The chance that a U or V value used in an H_2 query for an undefined sid is the same as one of the U or V values chosen when an honest party runs on that sid is at most $4/p$, where p is the order of the group. If $\mathcal{A}$ makes q_{H_2} H_2 queries, then the probability the simulation is invalid is at most $\Pr[\mathsf{E}_2] \leq 4q_{H_2}/p$.

Note that when a $\mathsf{Corrupt}$ query is made, the induced $\mathsf{GetTarget}$ query in $\mathrm{CPace}_{\mathrm{core}}$ causes the oracle H to be reprpgrammed, which in turn reprograms H_2. Thus we must consider the adversary's ability to notice such a reprogramming. However the only way to notice a reprogramming is to already have queried H_2 on a point that will induce a query to H on a reprogrammed point. The reprogrammed points are those which will eventually be used to induce session keys, queries of the form $H(\mathsf{ctr}, W_{i,0}^{u_i}, \mathrm{OC}(U_i, W_{i,0}))$ or $H(\mathsf{ctr}, W_{i,1}^{v_i}, \mathrm{OC}(V_i, W_{i,1}))$. But these are precisely the points that, if queried to H before a $\mathsf{Corrupt}$ query, then the reduction $\mathcal{B}$ wins the $\mathrm{CPace}_{\mathrm{core}}$ game. As a result, any advantage the adversary has in noticing reprogrammed points is exactly conferred to an advantage $\mathcal{B}$ has in winning $\mathrm{CPace}_{\mathrm{core}}$, and we need not be concerned with the probability this happens.

Assuming neither E_1 nor E_2 occur, there is no inconsistency in $\mathcal{B}$'s simulation of the BPR$'$ security game to $\mathcal{A}$. If $\mathcal{A}$'s output changes based on whether the answer to the Test query was real or random, then $\mathcal{B}$'s output will change based on whether the secret s in $\mathrm{CPace}_{\mathrm{core}}$ is 0 or 1. Combining these two statements yields

$$\mathrm{Adv}_{\mathrm{CPace}_{\mathrm{base}}}^{\mathrm{BPR}'}(\mathcal{A}) \leq \frac{4q_{H_2}}{p} + \mathrm{Adv}^{\mathrm{CPace}_{\mathrm{core}}}(\mathcal{B}) \ .$$

The runtime of $\mathcal{B}$ is the same as the runtime of $\mathcal{A}$, plus a small bookkeeping overhead for each query. Moreover, $\mathcal{B}$ makes the following number of queries to its $\mathrm{CPace}_{\mathrm{core}}$ oracles (distinguished with superscripts $\mathcal{B}$ and $\mathcal{A}$ when that oracle is available to both parties):

– $\cdot$ (the group operation oracle): $q_{\mathcal{G}}^{\mathcal{B}} = q_{\mathcal{G}}^{\mathcal{A}}$
– DLOG: $q_D^{\mathcal{B}} = q_D^{\mathcal{A}}$
– H: $q_H^{\mathcal{B}} \leq q_{H_2}^{\mathcal{A}}$
– GetGen: $q_G^{\mathcal{B}} \leq q_{H_1}^{\mathcal{A}}$
– $\mathsf{NewInstance}$: $q_N^{\mathcal{B}} \leq q_E^{\mathcal{A}} + q_S^{\mathcal{A}} + q_{Co}^{\mathcal{A}} + q_{H_1}^{\mathcal{A}}$
– $\mathsf{Challenge}$ queries of type 1: $q_{C1}^{\mathcal{B}} \leq q_E^{\mathcal{A}}$
– $\mathsf{Challenge}$ queries of type 2: $q_{C2}^{\mathcal{B}} \leq q_S^{\mathcal{A}}$
– $\mathsf{GetTarget}$: $q_T^{\mathcal{B}} \leq q_{Co}^{\mathcal{A}}$ □