Bare PAKE: Universally Composable Key Exchange from just Passwords

Manuel Barbosa, Kai Gellert, Julia Hesse*, and Stanislaw Jarecki

University of Porto (FCUP) and INESC TEC and Max Planck Institute for Security and Privacy
mbb@fc.up.pt

University of Wuppertal
kai.gellert@uni-wuppertal.de

IBM Research Europe - Zurich
jhs@zurich.ibm.com

UC Irvine
stanislawjarecki@gmail.com

Abstract. In the past three decades, an impressive body of knowledge has been built around secure and private password authentication. In particular, secure password-authenticated key exchange (PAKE) protocols require only minimal overhead over a classical Diffie-Hellman key exchange. PAKEs are also known to fulfill strong composable security guarantees that capture many password-specific concerns such as password correlations or password mistyping, to name only a few. However, to enjoy both round-optimality and strong security, applications of PAKE protocols must provide *unique* session and participant identifiers. If such identifiers are not readily available, they must be agreed upon at the cost of additional communication flows, a fact which has been met with incomprehension among practitioners, and which hindered the adoption of provably secure password authentication in practice.

In this work, we resolve this issue by proposing a new paradigm for truly *password-only* yet securely composable PAKE, called *bare* PAKE. We formally prove that two prominent PAKE protocols, namely CPace and EKE, can be cast as bare PAKEs and hence do not require pre-agreement of anything else than a password. Our bare PAKE modeling further allows to investigate a novel "reusability" property of PAKEs, i.e., whether n^2 pairwise keys can be exchanged from only n messages, just as the Diffie-Hellman non-interactive key exchange can do in a public-key setting. As a side contribution, this add-on property of bare PAKEs leads us to observe that some previous PAKE constructions relied on unnecessarily strong, "reusable" building blocks. By showing that "non-reusable" tools suffice for standard PAKE, we open a new path towards round-optimal post-quantum secure password-authenticated key exchange.

*The author was supported by the Swiss National Science Foundation (SNSF) under the AMBIZIONE grant "Cryptographic Protocols for Human Authentication and the IoT.

Table of Contents

1 Introduction

Passwords continue to be a dominant form of user authentication on the internet. While some alternatives to password-based authentication emerge, such as WebAuthn [W3C17], passwords remain prevalent, primarily due to usability issues (see e.g. [BHvS12]). More broadly, there is no alternative to low-entropy secrets in applications where authentication is based on *something you know*, such as a Personal Identification Number (PIN). Finally, passwords also play a central role at the infrastructure level, where they are widely used for device authentication, e.g., when setting up local wireless networks and IoT devices [HGP+18].

Due to its practical significance, cryptographically secure password authentication is an active area of research that dates back to the seminal paper of Bellovin and Merritt [BM92], more than thirty years ago. These protocols aim at ensuring *minimal leakage* of passwords while still allowing for checking their correctness. However, despite the impressive body of knowledge that was built during this time, there has not been much adoption of provably secure password authentication in the real world. Almost all password-protected login sites still deploy the privacy elusive “password-over-TLS”, where cleartext passwords are handed over to service providers, who hash the password and compare it with their database. For PIN-based authentication, ad-hoc solutions are also often deployed and are then subject to a-posteriori analyses (e.g., Password Authenticated Connection Establishment (PACE) [BFK09], and the Client-to-Authenticator Protocol in FIDO2 [BCZ22, BBCW21]).

One reason for the adoption of provably secure password authentication to remain scarce is that there are almost no *specifications* that could serve as a basis for implementors to adopt this type of cryptography. Writing good specifications is challenging, and in the case of password authentication, it is further hindered by a gap between the security models used to formally analyze such protocols and their use in practice. In particular, existing models [CHK+05, GMR06, DHP+18, Hes20, ABB+20] impose on the implementation a notion of a *unique party identifier* and/or a *unique session identifier*, and it is often unclear [Can19] how these parameters should be instantiated in practice. Known ways to correctly instantiate them impose significant protocol costs. The consequences are either specifications to which proofs in the literature do not apply or ones that are potentially unnecessarily inefficient.

In this paper, we revise models for secure password authentication to optimize for specification simplicity and implementation efficiency. In particular, we dispense with the need for the above-mentioned identifiers, introduce a security notion that captures true *password-only* authentication, and show that password-only implementations of known protocols meet this notion. Overall, we simplify and clarify the requirements that an implementer must meet to deploy password-authenticated protocols, leading to standards that are easier to adopt.

Security Models for Password Authentication. Cryptographic literature models password authentication as a Password-Authenticated Key Exchange

(PAKE) [BM92], which establishes a secure shared key between two participants if they run the protocol on the same password. PAKE security was first formalized in a game-based model by Bellare-Pointcheval-Rogaway (BPR) [BPR00]. The BPR model has been adopted in the analysis of many PAKE proposals, e.g., EKE [BPR00], SPEKE [Jab97, Mac01, HS14], SPAKE2 [AP05], TBPEKE [PW17], and CPace [AHH21]. However, the BPR model provides only limited assurance of real-world security, most importantly because it analyzes each user in separation, and assumes that every user chooses their password independently, while in the real world, password choices are often correlated, e.g. with prior passwords of the same user, or with passwords used by other family members, etc. To deal with these (and other) shortcomings of the BPR model, Canetti et al. [CHK$^+$05] proposed a Universally Composable (UC) PAKE model which addresses password correlations, password mistyping, password information leakage, and arbitrary interactions between protocol instances. The UC model further ensures security under arbitrary protocol composition, enabling security arguments for protocols that use generic PAKE as a subroutine, e.g., generic compilers from symmetric PAKE to asymmetric PAKE (where one party, the server, knows only a password hash instead of the password itself) [GMR06, HJK$^+$18], or from asymmetric PAKE to strong asymmetric PAKE (where the server's password hash is privately salted) [JKX18]. For these reasons the UC PAKE model has become a gold standard of PAKE security [GMR06, JKX18, BJX19, Hes20, ABB$^+$20, GJK21, AHH21, SGJK22, SGJ23].

Limitations of the UC PAKE model. Despite their widely recognized benefits, all previous UC PAKE models[1] come with hidden costs and fail to implement password-only authentication. Indeed, according to these models, each protocol participant $\mathcal{P}$ must supply three further inputs: i. a *session identifier sid* ii. a *party identifier*, and iii. a corresponding identifier $\mathcal{CP}$ of the intended counterparty in this protocol instance.[2]

The UC PAKE requirements on the session identifier are due to general conventions of the UC framework [Can01a], namely that each protocol instance must be identified by a globally unique session identifier. The standard UC PAKE model [CHK$^+$05] implies that PAKE participants can successfully establish a joint session key only if they use the same session identifier *sid*. Moreover, the requirement for global uniqueness implies no security guarantees to an honest party that re-uses the same *sid* string that was used by some pair of honest parties before. The requirements on party identifiers are similar: The UC framework assumes that each party that participates in the protocol has a unique identifier, i.e., that no two honest parties carry the same identifier $\mathcal{P}$.[3] The UC PAKE

[1] We use the term *standard UC PAKE* to denote the original notion of Canetti et al. [CHK$^+$05], and *UC PAKE* for including variants thereof, e.g., [Hes20, ABB$^+$20].

[2] The same problem appears in the BPR model for party identifiers, but not for session identifiers, which are protocol outputs rather than inputs.

[3] For a protocol realizing an ideal functionality that runs an independent session, with a globally unique session identifier, for a small set of parties, this means only that each of these participants must have a different party identifier. However, when

model enforces the use of these identifiers in the protocol because it allows the parties to establish a key only if they use matching identifiers, i.e., if one party uses its own and counterparty identifiers $\mathcal{P}$ and $\mathcal{CP}$, then the other must use respectively $\mathcal{P}'$,$\mathcal{CP}'$ s.t. $\mathcal{P}' = \mathcal{CP}$ and $\mathcal{CP}' = \mathcal{P}$.

Problems caused by requirements on sid,$\mathcal{P}$,$\mathcal{CP}$ **identifiers.** All of these inputs are problematic for an implementer, and they cannot be set to "don't care" symbols because of the above requirements of the UC PAKE model. Regarding the globally unique *sid* requirement, the standard way of satisfying it, as suggested in [Can01a], is to precede a PAKE instance by a *sid*-picking protocol, where *sid* is set as a concatenation of two fresh nonces which the parties send to each other. However, this adds a network flow to the resulting implementation. Anyone who attempts to implement a PAKE in any application would surely be annoyed by this requirement, in particular, because the only justification that researchers could give is that unique identifiers are required by the UC PAKE model. While there is no known attack on implementations that ignore the uniqueness of identifiers, or skip them completely, current proofs in the literature do not carry over to such more efficient and straightforward PAKEs.

The UC PAKE requirement that participants hold unique identifiers for themselves and their counterparties is also problematic. How should such identifiers be implemented? Should they be IP addresses? First, it is not clear what security guarantees the model implies if two honest parties happen to use the same self-identifier, and IP addresses can be dynamic. Second, if one party is behind a firewall then their counterparty will not know their IP address. Should the identifiers be DNS names instead? That is good for servers but not for clients. In summary, whatever an implementation chooses to use for these identifiers, it is possible that they turn out not to be unique, or it will tie the implementation too strongly to a particular networking setting, and it will break if the setting changes. In this aspect too, an implementer can ask why they need these unique identifiers at all as an input to the protocol, and a cryptographer's answer is currently the same as above, i.e. that current proofs do not imply that a PAKE is UC secure without them.

Given the troubles of pre-exchanging and agreeing on all these identifiers, the following research question repeatedly comes up while standardizing or implementing UC PAKE protocols:

Can PAKE protocols enjoy composable security without relying on pre-agreed unique session- and party identifiers?

Our proposal: The UC bare PAKE model. In this work, we answer the above question in the affirmative. We propose to shift from standard UC PAKE to *bare* PAKE (bPAKE), which models composable key exchange from *just* passwords. Our bPAKE model eliminates all the above implementation dilemmas regarding identifier choices, and consequently yields more practical protocols that

analyzing protocols that realize multi-instance versions of ideal functionalities that may interact with an arbitrary number of honest participants, the uniqueness of party identifiers becomes a global requirement.

do not require the application to pre-agree on anything other than a password: In our bare PAKE model the only required input *is* the password. Session identifiers still appear in our model, albeit only as an *output* of a protocol. They can be thought of (and used as such by applications) as a protocol transcript that uniquely identifies one particular key exchange session.

If two parties want to use PAKE to establish an agreement not only on their passwords but also on identifiers that might not be pre-agreed before the protocol starts, bare PAKE allows each party to enter its name as an optional input, and such name becomes part of their counterparty output. This way an application can reject a session if it was established with the counterparty whose name does not match the application's criteria, but these names do not have to be pre-agreed, and instead they can be communicated as part of the PAKE protocol. Furthermore, the name each party supplies is an arbitrary value, does not have to be unique, and can be set to $\perp$.

In summary, our UC bare PAKE functionality is just like the standard UC PAKE functionality in that it password-authenticates not only a session key but also a session and counterparty identifiers, but the latter are *protocol outputs instead of inputs*. The UC bare PAKE model therefore offers a simpler and application-friendly PAKE syntax—we argue that it is even friendlier than that of the BPR model [BPR00]—but it is a variant of the UC PAKE model of Canetti et al. [CHK$^+$05], hence it assures all the good properties of the UC framework including security under arbitrary password correlations, concurrent executions, and arbitrary protocol composition.

Further benefits and new insights: Reusability. The UC bare PAKE model draws further benefits from the above simplifications. If a PAKE protocol considers a password as the only necessary input, and the session identifier and counterparty's optional name are protocol outputs, then it becomes possible for a bare PAKE protocol participant to reuse their state and create keys with several counterparties from one password, where each key is accompanied by a separate session identifier and counterparty's name. Most PAKE schemes from the literature are password-encoded variants of the Diffie-Hellman (DH) non-interactive key exchange (NIKE) [DH76], which is known to enable the exchange of n^2 pairwise keys between n users. Can these PAKEs, such as EKE [BM92, DHP$^+$18] or CPace [HL17, AHH21], allow for a similar flavor of reusability, i.e., allow parties to reuse their state to produce *password-authenticated* keys from many incoming messages instead of just one? The standard UC PAKE versions of these protocols could not do that, because they require each party to specify a unique session and counterparty identifier as an input for each exchanged key. By contrast, bare PAKE allows us to investigate if PAKEs can be reusable.

As an application example, consider a client C who does not remember which password she should use with server S, but holds n password candidates, while server S has a single password registered for this user. With a reusable bare PAKE, C can run n PAKE instances, one for each password candidate, while S runs a single instance on its single password. If S's PAKE message is processed by each of C's instances, the client computes n session keys, and when S's instance processes each of the n messages sent by C's instances, the server also computes n session keys, and then they can learn if any of C's passwords is correct using a key confirmation protocol. One could use n independent instances of UC PAKE in the same application, but a reusable bare PAKE reduces the costs because S initializes only one PAKE instance and sends only one PAKE message to C.[4]

In another example, consider an Internet of Things (IoT) setting [Wik23], where each IoT device at home or office is initialized with the same password. With a reusable bare PAKE, each device can broadcast a single message to the network and then establish a password-authenticated pairwise key with any other device after processing that device's PAKE message. Concretely, if EKE or CPace are secure UC bare PAKEs, then all devices will be able to establish password-authenticated pairwise keys at the cost of a single broadcast per party!

Our Contributions. Besides the formal modeling of the new bare PAKE paradigm, our paper contains several contributions that we summarize below.

(1) The UC bare PAKE model. We define a UC bare PAKE model which reformulates the standard UC PAKE notion of Canetti et al. [CHK$^+$05] to make PAKEs truly *password-only*. A PAKE scheme secure in the UC bare PAKE model does not need pre-established unique session or party identifiers, which removes unnecessary hurdles for implementers. We stress that we achieve this without sacrificing *any* composability guarantees: our definitional work guarantees that the standard UC composition theorem implies that any usage of bare PAKE in an application can be replaced with the bare PAKE ideal functionality in the security analysis. Furthermore, the bare PAKE model allows UC PAKEs to be *reusable*, i.e., it allows for a single PAKE instance to establish password-authenticated session keys with an arbitrary number of parties. In our presentation we rely on the notion of structured UC protocols [Can01b, eprint 2020 version] to clarify the subtle syntactical issues of what it means for a UC protocol not to depend on the local party and session identifiers, even though these technical artifacts are inherent to a security analysis in the UC framework.

(2) Compilers. Notably, the input/output interfaces of our bare PAKE model and previous models differ, e.g., standard UC PAKE requires a new password input for each key exchange while bare PAKE can exchange many keys from one input password. It is hence not possible to say that the UC model of bare PAKE is stronger than the one for standard PAKE, or vice versa. To demonstrate the usefulness of our bare PAKE notions, we thus give generic and simple compilers between them. First, we show that any UC bare PAKE can be generically converted to a standard UC PAKE, by simply running the bare PAKE on $\bar{pw} = pw||sid||\mathcal{P}||\mathcal{CP}$. This compiler justifies our bare PAKE approach in the sense that our model is strong enough to realize standard PAKE, essentially without a performance penalty. The compiler further offers a smooth migration path towards the bare PAKE paradigm: applications that for some reason *do*

[4] This application appeared in [KM15], which give a BPR-model analysis of a solution assuming pseudorandomness of PAKE protocol messages. Note that S can limit guessing attempts by upper-bounding the number n of C's instances it processes.

have unique session- and party identifiers at hand get the expected standard PAKE interface from a bare PAKE as well! Second, we show that any secure standard UC PAKE implies a *one-time* UC bare PAKE, where one-time means that a party terminates a password instance after producing one key from it. The compiler lets parties exchange unique session- and party identifiers before executing the standard UC PAKE with them. In settings where no external uniqueness guarantee on party identifiers is available, our compiler reflects the overhead that implementations need to account for if only standard UC PAKE protocols are available.

(3) CPace is secure without identifiers, and reusable. We then investigate which PAKEs from the literature can be cast as bare PAKEs, i.e., can be deployed without unique session or party identifiers and establish keys from pre-agreed passwords alone. Of course, with our second compiler above, every standard UC PAKE can be converted into a bare PAKE, albeit at the cost of one additional round of nonce exchanges. We are instead looking for round-preserving bare versions of Diffie-Hellman-based PAKEs, i.e., that only take one round. We start with CPace [HL17], a PAKE that won the recent symmetric PAKE selection process [CFR20] of the IRTF and which is currently undergoing standardization [AHH20]. CPace lets parties perform a Diffie-Hellman key exchange with a group generator $g \leftarrow H(pw||sid||\mathcal{P}||\mathcal{CP})$ that encodes the password and all identifiers. We prove that "bare CPace" securely realizes our new notion with a generator $g \leftarrow H(pw)$ that just encodes the password. Our proof relies on the same assumptions taken in the analysis of its $(sid, \mathcal{P}, \mathcal{CP})$-dependent version [AHH21]. Our result implies reusability and composable security guarantees of the Internet-Draft version of CPace [AHH20] with $sid, \mathcal{P}, \mathcal{CP}$ all set to $\perp$. We recommend that the specification switches to bare CPace, as this would enable the authenticated transmission of party names without any uniqueness requirements.

(4) EKE is secure without identifiers, and reusable. We continue with the seminal Encrypted Key Exchange (EKE) protocol of Bellovin and Merrit [BM92, BPR00], which was the first in line of the many PAKE protocols built upon the non-interactive Diffie-Hellman key exchange. The idea of EKE is to password-encrypt the DH public keys, i.e., using a symmetric cipher and the password as the encryption key. In our work, we consider a general "bare EKE compiler" which starts from any non-interactive key exchange (NIKE) and any symmetric cipher, to obtain a PAKE that requires only pre-agreed passwords and doesn't need any additional $sid, \mathcal{P}, \mathcal{CP}$ inputs. We prove that this compiler can transform *any* NIKE that satisfies the standard notion of security for these protocols [CKS08, FHKP13] into a reusable bare PAKE in the ideal cipher (IC) model. Our result shows that EKE can enjoy composability without unique identifiers, and it can be used to exchange an arbitrary number of keys from just one password. As a corollary of this result we get a modular proof that *Hashed* Diffie-Hellman (Hashed DH) NIKE—i.e., a Diffie-Hellman NIKE where the pairwise key is computed as $K_{i,j} = H(g^{x_i \cdot x_j})$ where H is a hash function—can be

compiled into a *reusable* bare PAKE using EKE. The resulting protocol is tightly secure under gap-CDH in the Ideal Cipher and Random Oracle Models.

(5) Towards post-quantum secure PAKE from NIKE. An interesting side observation of our work on the bare version of EKE is that the reusability aspect crucially relies on the fact that secret keys of the underlying NIKE must be secure to reuse. Because standard UC PAKE does *not* allow for such reuse, an immediate question is whether we can relax the assumptions on the NIKE when aiming for the non-reusable standard UC PAKE notion. Indeed, we can show that a "one-time" version of NIKE is enough to realize standard UC PAKE and, furthermore, that "one-time" NIKE follows from passively secure NIKE in the Random Oracle Model. As a corollary, we derive that Hashed DH NIKE is one-time secure under CDH in the ROM, a result that explains previous CDH-based EKE security proofs [DHP$^+$18] with a nice flavor of modularity. These results also open a path for round-optimal post-quantum secure EKE instantiations based on, for example, the passively secure version of Swoosh [GdKQ$^+$23].

Other Related work. Shoup [Sho20] carries out an analysis of a concrete PAKE protocol in a variant of the UC model that addresses some of the concerns we also address in this paper. In particular, globally unique session identifiers are already replaced by locally unique instance identifiers, and session identifiers are seen as protocol outputs. However, the protocols in this model are still not password-only due to the use of party identifiers.

Küsters and Tuengerthal [KT11] identify conditions on protocols under which single-session security implies security in the *local-sid* multi-session setting. For the case of PAKE the latter is like our bPAKE functionality, except it still requires globally unique party identifiers and does not support re-use. However, their results assume multi-round protocols, since the simulator for the single-session protocol must compute each party's first message by following their protocol. Moreover, the joint-state cryptographic tools they consider do not include Ideal Ciphers or Random Oracles, which are essential in resp. EKE and CPace.

The NIST post-quantum competition and the upcoming standardization of Kyber KEM have driven several proposals of black-box constructions of PAKE protocols from KEM [BCP$^+$23, SGJ23]. The modularity of these constructions is aligned, and indeed closely related to, our analysis of EKE-NIKE. One substantial difference, however, is that we focus on reusable single-simultaneous-flow PAKEs, which we show to be easy to construct from simplified[5] NIKE. KEM-based constructions seem to allow only one-sided reusability by the party who generates the KEM public key; indeed, the other party is carrying out KEM encryption and stores no reusable state. We do not explore one-sided reusability in this paper, but our definitions already cater to this possibility for future work.

Dos Santos et al. [SGJ23] propose an alternative to the use of the Ideal Cipher in EKE variants that they call a *randomized Half-Ideal-Cipher*. This proposal solves an important problem in EKE, which is how to instantiate an Ideal Cipher

[5] In contrast to the standard notion of NIKE, where party identities are an input to the protocol, simplified NIKE has only public keys. [FHKP12]

over a structured data type such as a KEM public key, or a group element in DH-based systems. The Half-Ideal Cipher is essentially a two-round Feistel network, where a small part of the input (of size twice the security parameter) is generated uniformly at random and encrypted using an Ideal Cipher that works over this small domain. The remaining (potentially large) part of the input, e.g., the KEM public key, is simply masked using a group operation according to the structure of the Feistel construction. We believe that the proof of the EKE-NIKE construction we present in this paper can be easily adapted to rely on the same half-ideal cipher construction, thereby simplifying its instantiation with post-quantum secure NIKEs such as Swoosh [GdKQ$^+$23].

2 Preliminaries

Notation. We denote the security parameter as κ. We denote $[\ell] = \{1, \dots, \ell\}$. We write $x \leftarrow_{\mathrm{R}} \mathcal{S}$ for sampling x from uniform distribution over set $\mathcal{S}$. For a probabilistic polynomial-time (PPT) algorithm $\mathcal{A}$ we write $y \leftarrow_{\mathrm{R}} \mathcal{A}(x_1, \dots, x_\ell)$ for assigning to y an output of a randomized execution of $\mathcal{A}$ on input $x_1, \dots, x_\ell$. For any variable a, by *record* $\langle a, [b] \rangle$ we denote a 2-tuple of values, where the first value is equal to a, and we assign variable name b to the second entry. We use $\{a, b\}_{\mathsf{ord}}$ to denote string $a||b$ if $a \leq_{\mathsf{lex}} b$ and string $b||a$ otherwise.

2.1 Computational Assumptions

We recall several standard computational assumptions we use in our security proofs. The last assumption, sim-gapCDH is used in the security analysis of CPace, both by us and in prior works on CPace [HL17, AHH21].

Definition 1 (Pseudorandomness). *Let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to \{0,1\}^\kappa$ be an efficiently computable function. We call F* pseudorandom *if for all PPT $\mathcal{A}$ there exists a negligible function μ, such that for all $\kappa \in \mathbb{N}$ we have*

$$\mathbf{Adv}^{\mathsf{random}}_{\mathcal{A},F}(\kappa) := \left| \Pr_{k \leftarrow_R \{0,1\}^\kappa}[\mathcal{A}^{F(k,\cdot)}(1^\kappa) = 1] - \Pr_{f \leftarrow_R \mathsf{Func}_\kappa}[\mathcal{A}^{f(\cdot)}(1^\kappa) = 1] \right| \leq \mu(\kappa),$$

where Func_κ is the set of all functions of input and output length κ.

Definition 2 (Computational Diffie-Hellman (CDH)). *Let $\mathbb{G}$ be a family of cyclic groups indexed by the security parameter, each of some prime order q and with generator g. Then we say that the* Computational Diffie-Hellman assumption *holds in $\mathbb{G}$ if for every PPT adversary $\mathcal{A}$ there exists a negligible function μ, such that for all $\kappa \in \mathbb{N}$ it holds*

$$\mathbf{Adv}^{\mathsf{CDH}}_{\mathcal{A},\mathbb{G}}(\kappa) := \Pr\left[\mathcal{A}(\mathbb{G}_\kappa, g, g^a, g^b) = g^{ab} \mid a, b \leftarrow_R \mathbb{Z}_q\right] \leq \mu(\kappa).$$

Definition 3 (Gap Computational Diffie-Hellman (gapCDH)). *Let $\mathbb{G}$ be a family of cyclic groups, each with some prime order q. Then we say that* *the* Gap Computational Diffie-Hellman assumption *holds in* $\mathbb{G}$ *if for every PPT adversary* $\mathcal{A}$ *there exists a negligible function* μ*, such that for all* $\kappa \in \mathbb{N}$ *it holds that*

$$\mathbf{Adv}^{\mathsf{gapCDH}}_{\mathcal{A},\mathbb{G}}(\kappa) := \Pr\left[\mathcal{A}^{\mathsf{DDH}_x(\cdot,\cdot)}(\mathbb{G}, g, g^a, g^b) = g^{ab} \,\middle|\, \begin{array}{c} g \leftarrow_R \mathbb{G}_\kappa \\ a, b \leftarrow_R \mathbb{Z}_q \end{array}\right] \leq \mu(\kappa)\,,$$

where oracle $\mathsf{DDH}_x(Y, Z)$ *returns* 1 *if* $Z = Y^x$ *and* 0 *otherwise, for* $x = a, b$*.*

Definition 4 (Simultaneous Gap Computational Diffie-Hellman (sim-gapCDH)). *Let* $\mathbb{G}$ *be a family of cyclic groups, each with some order* q*. Then we say that the* Simultaneous Gap Computational Diffie-Hellman assumption *holds in* $\mathbb{G}$ *if for every PPT adversary* $\mathcal{A}$ *there exists a negligible function* μ*, such that for all* $\kappa \in \mathbb{N}$ *it holds that* $\mathbf{Adv}^{\mathsf{sim-gapCDH}}_{\mathcal{A},\mathbb{G}}(\kappa) :=$

$$\Pr\left[\mathcal{A}^{\mathsf{DDH}_x(\cdot,\cdot)}(\mathbb{G}_\kappa, g, g^r, g^{r'}, g^a) = (B, B^{a/r}, B^{a/r'}) \,\middle|\, \begin{array}{c} g \leftarrow_R \mathbb{G}_\kappa \\ a, r, r' \leftarrow_R \mathbb{Z}_q \end{array}\right] \leq \mu(\kappa),$$

where oracle $\mathsf{DDH}_x(Y, Z)$ *returns* 1 *if* $Z = Y^{a/x}$ *and* 0 *otherwise, for* $x = r, r'$*. In other words,* $\mathcal{A}$ *wins if it provides* B, K, K' *such that* $\mathsf{DDH}(g^r, g^a, B, K) = \mathsf{DDH}(g^{r'}, g^a, B, K') = 1$.

3 Bare Password-Authenticated Key Exchage

3.1 Previous UC PAKE models

We briefly recall the original UC PAKE model of [CHK+05] and its multi-session [CR03] and lazy-extraction [ABB+20] extensions, all captured in Figure 1, and then we introduce our UC Bare PAKE model, shown in Figure 2, and we explain its mechanics.

UC PAKE notion Canetti et al. The *single-session* PAKE functionality $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{ss}}}$ of [CHK+05], see Figure 1, defines how an ideal PAKE scheme should behave if each pair of participants uses a pre-agreed globally unique session identifier *sid* (and matching party and counterparty identifiers). The $\mathsf{NewSession}$ interface models the first party $\mathcal{P}$ using a given session identifier *sid* to initialize PAKE on password *pw*, with counterparty identifier $\mathcal{CP}$, and role specifying whether $\mathcal{P}$ plays an initiator or a responder role.[6] All inputs except for the password can be leaked, hence the ideal-world adversary $\mathcal{A}$ learns $(sid, \mathcal{P}, \mathcal{CP}, \mathsf{role})$. Functionality $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{ss}}}$ assumes that only two parties ever run the protocol using a given *sid*, and that these two parties respectively use identifiers $(\mathcal{P}, \mathcal{CP})$ and $(\mathcal{P}', \mathcal{CP}')$ s.t. $(\mathcal{P}', \mathcal{CP}') = (\mathcal{CP}, \mathcal{P})$, and any other $\mathsf{NewSession}$ requests are dropped.

When a session is initialized it is marked fresh, and the adversary has two options: (1) First, it can passively connect the two sessions using the matching

[6] Field role can be set to $\perp$ if the protocol is symmetric.

Session initialization, single session [CHK$^+$05]

On $(\mathsf{NewSession}, sid, \mathcal{P}, \mathcal{CP}, pw, \mathsf{role})$ from $\mathcal{P}$, send $(\mathsf{NewSession}, sid, \mathcal{P}, \mathcal{CP}, \mathsf{role})$ to $\mathcal{A}$, and if this is the first NewSession query, or it is the second one and record $\langle \mathcal{CP}, \mathcal{P}, [pw'] \rangle$ exists, then record $\langle \mathcal{P}, \mathcal{CP}, pw \rangle$ marked fresh.

Session initialization, multi session [CR03]

On $(\mathsf{NewSession}, sid, \mathcal{P}, ssid, \mathcal{CP}, pw, \mathsf{role})$ from $\mathcal{P}$ send $(\mathsf{NewSession}, sid, \mathcal{P}, ssid, \mathcal{CP}, \mathsf{role})$ to $\mathcal{A}$. If this is the first NewSession query for this $ssid$, or it is the second one and record $\langle \mathcal{CP}, ssid, \mathcal{P}, [pw'] \rangle$ exists, record $\langle \mathcal{P}, ssid, \mathcal{CP}, pw \rangle$ marked fresh.

Active attack

On $(\mathsf{TestPwd}, sid, \mathcal{P}, ssid, pw^*)$ from $\mathcal{A}$, if $\exists$ record $\langle \mathcal{P}, ssid, [\mathcal{CP}, pw] \rangle$ marked fresh then:

- If $pw^* = pw$ then change the mark to compromised and reply "correct guess".
- Otherwise change the mark to interrupted and reply "wrong guess".

Lazy password extraction [ABB$^+$20]

- On $(\mathsf{RegisterTest}, sid, ssid, \mathcal{P})$ from $\mathcal{A}$, if $\exists$ record $\langle \mathcal{P}, ssid, \dots \rangle$ marked fresh then change the mark to interrupted and flag it Tested;
- On $(\mathsf{LateTestPwd}, sid, ssid, \mathcal{P}, pw^*)$ from $\mathcal{A}$, if $\exists$ record $\langle \mathcal{P}, ssid, [\mathcal{CP}, pw, K] \rangle$ with flag Tested then remove this flag and return K' to $\mathcal{A}$ s.t.:
  - If $pw^* = pw$ then set $K' := K$ else pick $K' \leftarrow_{\mathrm{R}} \{0,1\}^\kappa$.

Key generation

On $(\mathsf{NewKey}, sid, \mathcal{P}, ssid, K^*)$ from $\mathcal{A}$, if $\exists$ record $rec = \langle \mathcal{P}, ssid, [\mathcal{CP}, pw] \rangle$ not marked completed then do:

1. If the record is compromised, ~~or either $\mathcal{P}$ or $\mathcal{CP}$ is corrupted,~~ then set $K := K^*$.
2. If rec is marked fresh and $\exists$ record $\langle \mathcal{CP}, ssid, \mathcal{P}, pw, [K'] \rangle$ marked completed which was marked fresh when $\mathcal{CP}$ output $(sid, ssid, K')$ then set $K := K'$.
3. In all other cases pick $K \leftarrow_{\mathrm{R}} \{0,1\}^\kappa$.

Finally, append K to rec, change its mark to completed, output $(sid, ssid, K)$ to $\mathcal{P}$.

Fig. 1. The original single-session PAKE functionality $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{SS}}}$ of Canetti et al. [CHK$^+$05], with single-use session identifiers sid, includes dashed boxes and excludes light gray parts. Crossed-out code corresponds to subsequent patches [AHH21]. Excluding dashed boxes and including light gray parts creates a *multi-session* variant of the same functionality [CR03], denoted $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{MS}}}$ (or $\mathcal{F}_{\mathsf{PAKE}}$ for short), with global session identifier sid and single-use subsession identifiers $ssid$. Adding dark gray parts to $\mathcal{F}_{\mathsf{PAKE}}$ defines $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{LE}}}$, which allows *lazy extraction* from active attacks [ABB$^+$20]. Adding the solid boxes defines a *leaky* variant of either functionality, which the default version excludes.

$sid, \mathcal{P}, \mathcal{CP}$ inputs by passing the message between them, which is modeled by a session-termination command $(\mathsf{NewKey}, sid, \mathcal{P}, \cdot)$ issued when $\mathcal{P}$ is fresh: If this is the first party to terminate then $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{SS}}}$ makes it output a random key K (step 3 in NewKey). If it is the second party to terminate then $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{SS}}}$ makes it output the same key if the two parties used equal passwords (step 2), or an independent key if the two passwords were unequal (step 3). Alternatively, (2) the adversary can mount an active attack against either session, modeled by the $\mathsf{TestPwd}$ interface, which restricts each such attack to a unique password guess pw^*. If this guess is correct, the session's mark is changed to $\mathsf{compromised}$,[7] otherwise it becomes $\mathsf{interrupted}$.[8] Moreover, if $\mathcal{P}$'s session is $\mathsf{compromised}$ then $\mathcal{A}$'s session-termination query $(\mathsf{NewKey}, \mathcal{P}, sid, K^*)$ causes $\mathcal{P}$ to output an adversarially chosen key K^* (step 1), while if it is $\mathsf{interrupted}$ then $\mathcal{P}$ outputs a random key (step 3), but unlike in option (1) this key cannot be transferred to party $\mathcal{P}'$ because $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{SS}}}$ ensures that two parties can output the same key only if they use the same passwords *and* both terminate as fresh (see step 2).

Multi-session and lazy-extraction extensions of UC PAKE. Canetti and Rabin [CR03] showed that each single-session functionality can be cast as a multi-session one, where individual protocol instances are differentiated by unique *sub-session* identifiers $ssid$, and the session identifier sid designates global parameters e.g. a CRS, or an instance of a "globally available" functionality which all parties can rely on, e.g. a Random Oracle hash or an Ideal Cipher encryption. In Figure 1 we show this multi-session form of the UC PAKE functionality, denoted $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{MS}}}$. As shown in [CR03], if a protocol realizes $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{SS}}}$ then it also realizes $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{MS}}}$, as long as the environment ensures that tuples $(ssid, \mathcal{P}, \mathcal{CP})$ satisfy the same requirements posed above on tuples $(sid, \mathcal{P}, \mathcal{CP})$. In the remainder of this work we will treat the multi-session version of the PAKE functionality, i.e. $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{MS}}}$, as the definition of UC PAKE, denoted $\mathcal{F}_{\mathsf{PAKE}}$ for short.

Abdalla et al. [ABB$^+$20] introduced a *lazy-extraction* extension of the UC PAKE functionality, which we show as functionality $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{LE}}}$ in Figure 1. In this extension the adversary can use interface $\mathsf{RegisterTest}$ to actively attack a session on a committed but hidden password guess. Such session becomes $\mathsf{interrupted}$ and outputs a random key K when it is terminated via the NewKey query, but the adversary can then reveal a unique password guess pw^* used in this attack via interface $\mathsf{LateTestPwd}$, and $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{LE}}}$ responds with the correct key K if the guess is correct, or an independent value if the guess is wrong. We include this extension because [ABB$^+$20] showed it is necessary and sufficient to capture several round and computation-efficient PAKE protocols, including CPace [HL17, AHH21], which in this work we generalize to the UC bare PAKE model.

[7] In [CHK$^+$05] the PAKE functionality leaks if this case occures to the adversary, but our default notion omits that leakage since it is not present in the protocols we analyze.

[8] Either mark prevents $\mathcal{A}$ from issuing another $\mathsf{TestPwd}$ query for the same session.

3.2 The UC Bare PAKE model

In Figure 2 we describe an ideal functionality $\mathcal{F}_{\mathsf{bPAKE}}$ which defines our proposed UC *bare* PAKE (bPAKE) model. It is a multi-session functionality, where identifier *sid* identifies global common parameters, e.g. an instance of an Ideal Cipher or a Random Oracle. However, our bPAKE functionality $\mathcal{F}_{\mathsf{bPAKE}}$ makes several major departures from the multi-session PAKE functionality $\mathcal{F}_{\mathsf{PAKE}}$. First and foremost, functionality $\mathcal{F}_{\mathsf{bPAKE}}$ models PAKE as a *password-only* protocol, where the functionality does not enforce that the participants have pre-agreed unique subsession identifiers *ssid* or party identifiers $\mathcal{P}, \mathcal{CP}$. Indeed, interface $\mathsf{NewSession}$ in $\mathcal{F}_{\mathsf{bPAKE}}$ does not take inputs $(\mathcal{P}, \mathit{ssid}, \mathcal{CP})$ which were present in $\mathcal{F}_{\mathsf{PAKE}}$, and when sessions are passively connected, via interface $\mathsf{PassiveNewKey}$, functionality $\mathcal{F}_{\mathsf{bPAKE}}$ allows two sessions to output the same session key K only if their passwords *pw* are the same (see step 2 in $\mathsf{PassiveNewKey}$ in Fig. 2).

In addition to the password input *pw*, the other inputs of bPAKE (see the $\mathsf{NewSession}$ interface in Fig. 2) include an instance identifier i, and fields *id* and role. Input role plays the same function of distinguishing protocol initiators and responders as in UC PAKE. The *instance identifier* i plays no security role, and is used only to distinguish between many bPAKE instances which one party can execute, hence i's must only be locally unique. [9] Finally, *id* is a *party identifier* which party $\mathcal{P}$ uses in that instance. It is an arbitrary string, might not be unique, can be set to $\perp$, and $\mathcal{P}$'s counterparty does not need prior knowledge of it. Its security property is that if the PAKE instances of $\mathcal{P}$ and $\mathcal{P}'$ are passively connected then $\mathcal{P}$ outputs *cpid* used by $\mathcal{P}'$ as its counterparty name *cpid*, and vice versa. In particular, in bPAKE party $\mathcal{P}$ outputs an identifier of its counterparty instead of specifying that identifier in its inputs.[10]

The bPAKE functionality supports *(sub)session identifiers ssid*, but it treats *ssid* also as a protocol output rather than input. The session identifiers *ssid* are public and their only semantic implication is that each *ssid* output is globally unique except for a pair of instances which are passively connected. Functionality $\mathcal{F}_{\mathsf{bPAKE}}$ enforces this by storing previously used *ssid*'s in set S, and ensuring that any new *ssid* satisfies $\mathit{ssid} \notin \mathsf{S}$.[11] Note that this implies that each session key corresponds to a globally unique public *ssid*, and that different *ssid*'s are associated with independent random session keys. In more details, let $\mathcal{P}_i$ and $\mathcal{P}'_j$ be two bPAKE instances which are initialized by $\mathcal{P}$ and $\mathcal{P}'$ on respective inputs $(i, pw, id, \mathsf{role})$ and $(j, pw', id', \mathsf{role}')$, and which are passively connected by the adversary, and let $(K, \mathit{ssid}, \mathit{cpid})$ and $(K', \mathit{ssid}', \mathit{cpid}')$ be their outputs. Rules of $\mathcal{F}_{\mathsf{bPAKE}}$ imply that if $pw \neq pw'$ or $\mathit{ssid} \neq \mathit{ssid}'$ then K and K' are

[9] The model can be extended to allow $\mathcal{P}$ to terminate i-th instance and re-use the same i for a new one, and we omit this only for the sake of notational simplicity.

[10] Note that if a PAKE application wants its counterparty identifier to satisfy some access control predicate then it can apply it to the identifier *cpid* specified in the session outputs, and reject the session if that output fails to satisfy the predicate.

[11] $\mathcal{F}_{\mathsf{bPAKE}}$ rules allow the ideal-world adversary to set *ssid*'s at will when a session terminates, but each of our protocols implements *ssid* as a protocol transcript, and the global *ssid* uniqueness is assured by the entropy of protocol messages.

```
Functionality initiation: set $\mathsf{S} := \{\}$.

Session initiation

On $(\mathsf{NewSession}, sid, i, pw, id, \mathsf{role})$ from $\mathcal{P}$, send $(\mathsf{NewSession}, sid, \mathcal{P}, i, id, \mathsf{role})$ to
$\mathcal{A}$. If record $\langle \mathcal{P}, i, ... \rangle$ does not exists, record $\langle \mathcal{P}, i, pw, id \rangle$.

Key generation

On $(\mathsf{ActiveNewKey}, sid, \mathcal{P}, i, pw^*, K^*, ssid, cpid)$ from $\mathcal{A}$, if $\exists$ record $\langle \mathcal{P}, i, [pw, id] \rangle$:
  – // Repeat $ssid$ means repeat output.
    If $\exists$ record $\langle \mathsf{ses}_{\mathsf{act}}, \mathcal{P}, i, ssid, cpid, [K] \rangle$ output $(sid, i, K, ssid, cpid)$ to $\mathcal{P}$.
  – // Actively attacked parties still get $ssid$ uniqueness guarantee.
    Otherwise, if $ssid \notin \mathsf{S}$:
      • Add $ssid$ to $\mathsf{S}$
      • If $pw^* = pw$ then set $K := K^*$ else pick $K \leftarrow_{\mathrm{R}} \{0,1\}^\kappa$
      • If $pw^* = \perp$ then save $\langle \mathsf{latetest}, \mathcal{P}, ssid, pw, K \rangle$
      • Save $\langle \mathsf{ses}_{\mathsf{act}}, \mathcal{P}, i, ssid, cpid, K \rangle$
      • Output $(sid, i, K, ssid, cpid)$ to $\mathcal{P}$.

On $(\mathsf{PassiveNewKey}, sid, \mathcal{P}, i, \mathcal{P}', i', ssid)$ from $\mathcal{A}$, if $\exists$ record $\langle \mathcal{P}, i, [pw, id] \rangle$:
  – // Repeat $ssid$ means repeat output.
    If $\exists$ rec. $\langle \mathsf{ses}_{\mathsf{hbc}}, \mathcal{P}, i, [\mathcal{P}', i'], ssid, [cpid, K] \rangle$ output $(sid, i, K, ssid, cpid)$ to $\mathcal{P}$.
  – // If peer was not created, request is ignored.
    Otherwise, if $\exists$ record $\langle \mathcal{P}', i', [pw', id'] \rangle$ then set $cpid := id'$ and do:
      1. // Complete protocol for the first-to-terminate participant.
         If $ssid \notin \mathsf{S}$:
           Add $ssid$ to $\mathsf{S}$
           Pick $K \leftarrow_{\mathrm{R}} \{0,1\}^\kappa$.
           Save $\langle \mathsf{ses}_{\mathsf{hbc}}, \mathcal{P}, i, \mathcal{P}', i', ssid, cpid, K \rangle$
           Output $(sid, i, K, ssid, cpid)$ to $\mathcal{P}$.
      2. // Complete protocol for the second-to-terminate participant.
         If $\exists$ record $\langle \mathsf{ses}_{\mathsf{hbc}}, \mathcal{P}', i', \mathcal{P}, i, ssid, id, [K'] \rangle$:
           If $pw' = pw$ then set $K := K'$ else pick $K \leftarrow_{\mathrm{R}} \{0,1\}^\kappa$.
           Save $\langle \mathsf{ses}_{\mathsf{hbc}}, \mathcal{P}, i, \perp, \perp, ssid, cpid, K \rangle$
           Output $(sid, i, K, ssid, cpid)$ to $\mathcal{P}$.
      3. // Re-use of $ssid$ but with mismatched remaining data
         Otherwise, ignore $\mathsf{PassiveNewKey}$ request

Late Password Test Attack

On $(\mathsf{LateTestPwd}, sid, \mathcal{P}, i, ssid, pw^*)$ from $\mathcal{A}$, if $\exists$ record $\langle \mathsf{latetest}, \mathcal{P}, ssid, [pw, K] \rangle$
then delete this record and return $K'$ to $\mathcal{A}$ s.t.:
  – If $pw^* = pw$ then set $K' := K$ else pick $K' \leftarrow_{\mathrm{R}} \{0,1\}^\kappa$.
```

Fig. 2. The *bare* PAKE functionality $\mathcal{F}_{\mathsf{bPAKE}}$. Including dark grey text defines a *lazy extraction* version of this functionality, denoted $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$.

independent. Indeed, unless two random keys picked by $\mathcal{F}_{\mathsf{bPAKE}}$ collide, two passively connected instances output the same keys if and only if they use the same passwords and they are passively connected. Furthermore, if $\mathit{ssid} = \mathit{ssid}'$ then $(\mathit{cpid}, \mathit{cpid}') = (\mathit{id}', \mathit{id})$, i.e. shared ssid implies correct counterparty names. Taken together, $\mathcal{F}_{\mathsf{bPAKE}}$ rules imply that event $K = K'$ can occur if and only (except for negligible probability of collision among random keys) if either of the following two conditions hold:

1. $(\mathit{pw}, \mathit{ssid}, \mathit{id}, \mathit{cpid}) = (\mathit{pw}', \mathit{ssid}', \mathit{cpid}', \mathit{id}')$, moreover this case can occur only if $\mathcal{P}_i$ and $\mathcal{P}'_j$ are passively connected.
2. Both $\mathcal{P}_i$ and $\mathcal{P}'_j$ are actively attacked, via the $\mathsf{ActiveNewKey}$ interface, and both attacks are successful, i.e. the attack against $\mathcal{P}_i$ used $\mathit{pw}^* = \mathit{pw}$ and the attack against $\mathcal{P}'_j$ used $\mathit{pw}^* = \mathit{pw}'$. In this case $\mathit{cpid}, \mathit{cpid}'$ are arbitrary but $\mathit{ssid}, \mathit{ssid}'$ are globally unique, i.e. $\mathit{ssid} \neq \mathit{ssid}'$ and ssid and ssid' are not output as session identifiers on any other session (by any party).

Functionality $\mathcal{F}_{\mathsf{PAKE}}$ guarantees item (1) with regards to passively connected parties, but it requires $\mathit{ssid}, \mathit{id}, \mathit{cpid}$ to be pre-agreed, it requires $\mathit{id}, \mathit{cpid}$ (denoted $\mathcal{P}, \mathcal{P}'$ therein) to be globally unique, and it requires the environment to create a globally unique ssid, while $\mathcal{F}_{\mathsf{bPAKE}}$ does not impose such requirements on inputs, and instead enforces property (1) on its outputs. Moreover, functionality $\mathcal{F}_{\mathsf{PAKE}}$ does *not* enforce guarantee (2), in particular $\mathit{ssid} = \mathit{ssid}'$ can hold in all attack cases because these are $\mathcal{F}_{\mathsf{PAKE}}$ inputs instead of outputs.

The global uniqueness of ssid's implies that ssid output by $\mathcal{F}_{\mathsf{bPAKE}}$ is a *channel binder*, see e.g. [HJKW23], which uniquely identifies a channel defined by key K, and which can be used to authenticate it with secondary means, e.g. with a public key, using the SIGMAC method [Kra03, HJKW23]. By contrast, the fact that $\mathcal{F}_{\mathsf{PAKE}}$ does not enforce guarantee (2) implies that $\mathcal{F}_{\mathsf{PAKE}}$'s ssid's cannot play that role.[12]

Non-leaky and "free dating" aspects of bare PAKE functionality. Functionality $\mathcal{F}_{\mathsf{bPAKE}}$ handles active attacks and session outputs differently from $\mathcal{F}_{\mathsf{PAKE}}$. First, note that if functionality $\mathcal{F}_{\mathsf{PAKE}}$ is *non-leaky*, i.e. Fig. 1 omitting the fragments in solid boxes, the adversary $\mathcal{A}$ doesn't learn if $\mathit{pw}^* = \mathit{pw}$ in an active attack, therefore w.l.o.g. $\mathcal{A}$ can postpone an attack until the creation of the session output. This is why in $\mathcal{F}_{\mathsf{bPAKE}}$ we combine an active attack and session-output fixing into query $\mathsf{ActiveNewKey}$ that takes both a password pw^* and a key K^*. This has the same effect as sending $\mathsf{TestPwd}$ with pw^* followed by NewKey with K^* to $\mathcal{F}_{\mathsf{PAKE}}$. On the other hand, if a session concludes without an active attack, $\mathcal{A}$ can emulate that using query $\mathsf{PassiveNewKey}$.

Another $\mathcal{F}_{\mathsf{bPAKE}}$ feature is that the pair of protocol instances which can be *matched*, i.e. which share a key if they are passively connected and use the same passwords, is not fixed before the protocol starts, and instead can be

[12] The protocols of [GMR06, HJK$^+$18] run PAKE as subprotocol and need a binder to a PAKE-created key. To support this [GMR06] extended functionality $\mathcal{F}_{\mathsf{PAKE}}$ to export protocol transcript as such binder. The ssid output by $\mathcal{F}_{\mathsf{bPAKE}}$ can support the same function.

adaptively chosen in protocol execution. In $\mathcal{F}_{\mathsf{PAKE}}$ instances can match only if they use the same *ssid*, and the environment can give the same *ssid* to at most two instances. By contrast, matching in $\mathcal{F}_{\mathsf{bPAKE}}$ is based only on passwords, and a given instance can be potentially matched with any other which uses the same password. Hence query PassiveNewKey identifies the target instance $\mathcal{P}_i$ together with the counterparty instance $\mathcal{P}'_{i'}$ it connects with. If $id[\mathcal{P}_i]$ is the name used by instance $\mathcal{P}_i$ in NewSession, functionality $\mathcal{F}_{\mathsf{bPAKE}}$ reacts to $(\mathsf{PassiveNewKey}, \mathcal{P}, i, \mathcal{P}', i', \mathit{ssid})$ by picking random key K, setting $\mathit{cpid} := id[\mathcal{P}'_{i'}]$, storing $\langle \mathsf{ses}_{\mathsf{hbc}}, \mathcal{P}, i, \mathcal{P}', i', \mathit{ssid}, \mathit{cpid}, K\rangle$ and sending $(K, \mathit{ssid}, \mathit{cpid})$ to P_i. The adversary then has an option to also connect $\mathcal{P}'_{i'}$ to $\mathcal{P}_i$ by querying $(\mathsf{PassiveNewKey}, \mathcal{P}', i', \mathcal{P}, i, \mathit{ssid})$ with the same session identifier *ssid*. (If *ssid* is a protocol transcript this happens if the last message of $\mathcal{P}_i$ is sent to $\mathcal{P}'_{i'}$.) Since *ssid* is not new, i.e. $\mathit{ssid} \in \mathsf{S}$, and $\exists$ record $\langle \mathsf{ses}_{\mathsf{hbc}}, \mathcal{P}, i, \mathcal{P}', i', \mathit{ssid}, id[\mathcal{P}'_{i'}], K\rangle$, instance $\mathcal{P}'_{i'}$ will output $(K', \mathit{ssid}, \mathit{cpid}' = id[\mathcal{P}_i])$ for $K' = K$ if $pw[\mathcal{P}_i] = pw[\mathcal{P}'_{i'}]$ and random K' otherwise.[13]

Re-use of bare PAKE instances. If instance-matching in bPAKE is adaptive, then can it happen more than once? Indeed, if a PAKE protocol each party sends only one message,[14] party $\mathcal{P}_i$ sends its message m_i and holds a local state st_i, and when it receives message m_j from counterparty $\mathcal{P}_j$, it uses (st_i, m_j) to compute session output $(K, \mathit{ssid}, \mathit{cpid})$. But could $\mathcal{P}_i$ not terminate at that point, continue holding state st_i, and when $\mathcal{P}_i$ receives a second message, let's say message m_t from party $\mathcal{P}_t$, then could $\mathcal{P}_i$ compute another session output $(K', \mathit{ssid}', \mathit{cpid}')$, this time on input (st_i, m_t)?

Functionality $\mathcal{F}_{\mathsf{bPAKE}}$ allows for such re-use of bPAKE session state: After bPAKE instance $\mathcal{P}_i$ is created via NewSession, and e.g. if $\mathcal{P}_i$ is an initiator a real-world $\mathcal{P}_i$ would send its first message, every time a real-world adversary forwards to $\mathcal{P}_i$ a message from some other session $\mathcal{P}'_{i'}$, the ideal-world adversary $\mathcal{A}$ sends $(\mathsf{PassiveNewKey}, \mathcal{P}, i, \mathcal{P}'_{i'}, \mathit{ssid})$ with some *ssid*, and this query is processed as explained above, but this can happen unlimitted number of times for the same $\mathcal{P}_i$. Likewise, if the real-world adversary sends his own message instead of one produced by some honest instance, its ideal-world counterpart $\mathcal{A}$ will send $(\mathsf{ActiveNewKey}, \mathcal{P}, i, pw^*, K^*, \mathit{ssid}, \mathit{cpid})$, and since some messages the real-world $\mathcal{P}_i$ can be forwarded from honest parties and some created by the adversary, the ideal-world adversary $\mathcal{A}$ can send messages of the form $(\mathsf{PassiveNewKey}, \mathcal{P}, i, ...)$ and messages of the form $(\mathsf{ActiveNewKey}, \mathcal{P}, i, ...)$ interspered in an arbitrary sequence. Each command creates a new session information on $\mathcal{P}_i$, each one identified by a unique *ssid*: Some of them can represent passively connected sessions, some actively attacked ones, but $\mathcal{F}_{\mathsf{bPAKE}}$ rules imply that there is no

[13] See step (2) in PassiveNewKey in Figure 2, although it can be difficult to pattern-match the above with Figure 2 because in this second PassiveNewKey instances $\mathcal{P}'_{i'}$ and $\mathcal{P}_i$ play the opposite roles compared to the notation in that step in the figure.

[14] As e.g. in EKE [BPR00], SPEKE [Jab97, Mac01, HS14], SPAKE2 [AP05], TBPEKE [PW17], and CPace [AHH21].

difference between re-using the state of a single bPAKE instance $\mathcal{P}_i$ and running multiple independent PAKE instances on the same password. [15]

Finally, if $\mathcal{P}_i$ reuses its bPAKE state to process responses from multiple counterparties, then $\mathcal{P}_i$ might process the same response twice. Since this would make $\mathcal{P}_i$'s transcript the same in these two interactions, this corresponds to $\mathcal{A}$ re-using the same *ssid* in two key-generation queries, either PassiveNewKey or ActiveNewKey. In that case real-world $\mathcal{P}_i$ has identical session outputs in response to such queries, hence $\mathcal{F}_{\mathsf{bPAKE}}$ assures the same happens in the ideal world, by saving respectively $\langle \mathsf{ses}_{\mathsf{hbc}}, \mathcal{P}, i, ..., \mathit{ssid}, \mathit{cpid}, K\rangle$ or $\langle \mathsf{ses}_{\mathsf{act}}, \mathcal{P}, i, \mathit{ssid}, \mathit{cpid}, K\rangle$, and whenever PassiveNewKey and ActiveNewKey query is made for some $\mathcal{P}, i, \mathit{ssid}$, $\mathcal{F}_{\mathsf{bPAKE}}$ checks if the corresponding record exists, and if so then $\mathcal{F}_{\mathsf{bPAKE}}$ resends to $\mathcal{P}_i$ tuple $(K, \mathit{ssid}, \mathit{cpid})$ stored in that record.

3.3 Syntax of Bare PAKE and Structured Protocols

The way we defined $\mathcal{F}_{\mathsf{bPAKE}}$ allows us to think of a unique global instance of the bare PAKE ideal functionality/protocol to which a party can resort at any time to establish a key with a counterparty that uses the same password. Indeed, a party can create an arbitrary number of sessions using the input parameter i, and these will behave independently of each other. In what follows we discuss the syntax of such protocols in the real-world and clearly separate what is the code of a bare PAKE protocol from the boiler-plate infrastructure that manages communications, network addresses and other pieces of information that have no bearing on correctness and security.

A *structured protocol* [Can01b, 2020 version on eprint, Section 5] consists of a protocol *shell* and a protocol *body*. We leverage this terminology to resolve the conundrum of formalizing interactive protocols where message recipients are unknown: as proposed by Canetti, we use this formalism to syntactically restrict real-world protocol access to session identifiers and party identifiers that are a UC framework artifact and should not be required by the practical protocol. The idea is to let a shell, which is a modelling component in the security analysis that abstracts the operation of a maliciously controlled network, to manage the sending of outgoing messages and the assignment of incoming ones. The body runs the "cryptographic core", i.e., it executes the bare PAKE code oblivious of any addressing information and session identification that could be provided from the outside. In this section we give only the minimum terminology that should allow the reader to follow the rest of the paper, without being overwhelmed with the details. For completeness, Appendix A contains the full terminology and reasoning underlying structured protocols.

[15] If party $\mathcal{P}_i$ wants to use state st_i to process only n sessions then it can process only the first n session triples output by $\mathcal{P}_i$. This is equivalent to terminating a bPAKE instance, and one can extend $\mathcal{F}_{\mathsf{bPAKE}}$ to explicitly support such feature.

The Body: Syntax of a Bare PAKE. A bare PAKE protocol is a five-tuple of ppt algorithms (Setup, StartIni, StartRsp, EndIni, EndRsp).[16]

- Setup(1^κ) takes as input the security parameter and produces some parameters *prm* that are (locally) shared between multiple protocol instances.
- StartIni(prm, pw, id) takes as input the parameters *prm*, a password *pw* and a party name *id* and it outputs a first message m_1 and a state st_I.
- StartRsp(prm, pw, id) takes as input the parameters *prm*, a password *pw* and a party name *name* and it outputs a state st_R.
- EndRsp(st_R, m_1) takes as input a state st_R and a message m_1 and, if it completes successfully, it outputs a key K, a session identifier *ssid*, a counterparty name *cpid* and a message m_2. Otherwise it outputs $\perp$.
- EndIni(st_I, m_2) takes as input a state st_I and a message m_2 and, if it completes successfully, it outputs a key K, a subsession identifier *ssid* and a counterparty name *cpname*. Otherwise it outputs $\perp$. This algorithm is deterministic.

In the case of single-simultaneous-flow (SSF) protocols, only StartIni and EndIni need to be specified, and we assume that m_1 produced by StartIni can be used as an input to EndIni.

Correctness of the protocol requires that an honest execution results in both parties agreeing on the the same key K, session identifier *ssid* and obtain the counter-party name that was initially provided by their peer. Formally, the following should hold for all id_I, id_R, pw:

$$\Pr\left[\begin{array}{c|l} & prm \leftarrow_{\mathrm{R}} \mathrm{Setup}(1^\kappa) \\ (id_I, cpid_I) = (cpid_R, id_R) & (m_1, st_I) \leftarrow_{\mathrm{R}} \mathrm{StartIni}(prm, pw, id_I) \\ \wedge & st_R \leftarrow_{\mathrm{R}} \mathrm{StartRsp}(prm, pw, id_R) \\ (ssid_I, K_I) = (ssid_R, K_R) & (K_I, ssid_I, cpid_I, m_2) \leftarrow_{\mathrm{R}} \mathrm{EndRsp}(m_1, st_R) \\ & (ssid_R, cpid_R, K_R) \leftarrow \mathrm{EndIni}(m_2, st_I) \end{array}\right] = 1\,.$$

The Shell: handling instances and communications. The above body of a bare PAKE is completed with a "wrapper" that models communication and session handling, called a *shell*. The shell resolves questions such as *"Which of Alice's passwords is used to compute a key from an incoming message?"*. Various such wrappers could be defined: one that always compute the maximum number of keys from incoming messages, or one that allows pointing at a particular password instance to compute a key with. In this work, we mostly work with the latter option, as we believe most applications will be working in this scenario. We call this shell $\pi_{\mathrm{Sh}}$and formally state its code in Figure 13. For the sake of this overview, we illustrate its workings in Figure 3. π_{Sh} propagates messages to the "malicious network", which in UC terminology is represented by the adversary. This captures the non-determinism of where messages are sent to, i.e., all possible scenarios of who is using a message that the shell propagates.

[16] This limits our analysis to two-pass protocols, which we do in this paper for the sake of simplicity. Our approach can be extended to protocols with additional rounds.

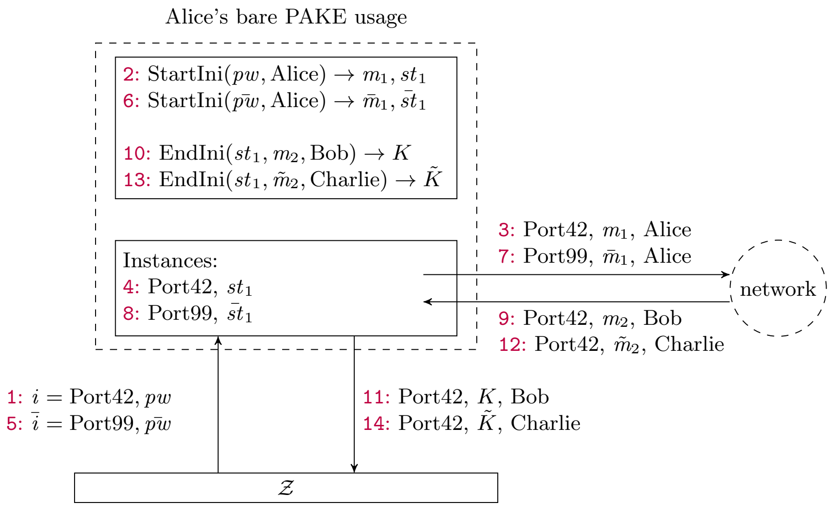

Alice's bare PAKE usage
2: StartIni(pw, Alice) → m_1, st_1
6: StartIni($\bar{pw}$, Alice) → $\bar{m}_1, \bar{st}_1$
10: EndIni(st_1, m_2, Bob) → K
13: EndIni($st_1, \tilde{m}_2$, Charlie) → $\tilde{K}$
Instances:
4: Port42, st_1
8: Port99, $\bar{st}_1$
3: Port42, m_1, Alice
7: Port99, $\bar{m}_1$, Alice
network
9: Port42, m_2, Bob
12: Port42, $\tilde{m}_2$, Charlie
1: i = Port42, pw
5: $\bar{i}$ = Port99, $\bar{pw}$
11: Port42, K, Bob
14: Port42, $\tilde{K}$, Charlie
$\mathcal{Z}$

Fig. 3. Illustration of the shell π_{Sh} handling multiple passwords of Alice (omitting interface names and session- and subsession identifiers from inputs, outputs and messages for brevity, and showing a single-simultaneous flow protocol). Upon NewSession input (1:) with instance identifier Port42 and password pw, the shell (2:) calls the body to produce a message, (3:) propagates it to the network and stores the instance state (4:). The same happens for another input password $\bar{pw}$ with instance identifier Port99 (5:,6:,7:,8:). An incoming message (9:) is inspected by the shell for its instance identifier. The shell finds a corresponding instance, (10:) invokes the body with the incoming message and the instance state and (11:) outputs the key. Another incoming message (12:) for the same instance is treated the same way (13:,14:), resulting in Alice sharing keys with Bob and Charlie on instance Port42.

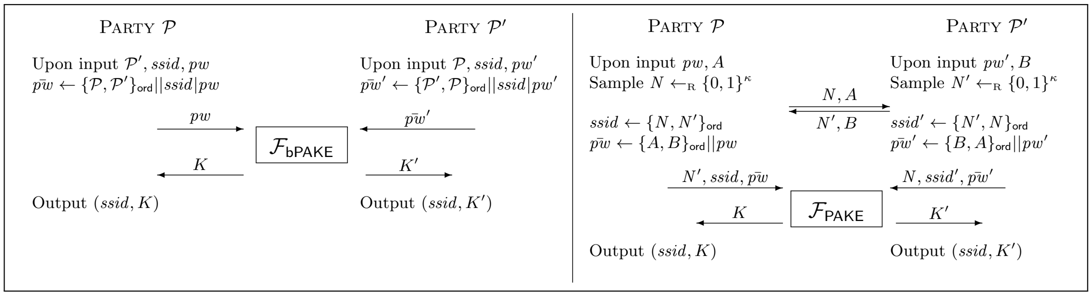
4 Transformations between PAKE and Bare PAKE

From a theoretical point of view, one might ask how the two primitives relate, e.g., "Is PAKE stronger than bPAKE?" or vice versa, but one cannot say that either functionality directly implies the other in the UC-emulation sense, since their input/output behaviors are trivially distinguishable. Instead, we consider more interesting practical questions of whether we can build one protocol from the other:

- Suppose we have an implementation of a UC PAKE, but we do not know how to guarantee global session identifier uniqueness, nor are we sure what kind of party identifiers to use. Can we use it to agree on a key based only on a password? In other words, can we build a bare PAKE from a PAKE?
- Conversely, suppose we have an implementation of a bare PAKE, and we want to integrate it into a higher level protocol that expects the PAKE interface. Can we build a PAKE from a bare PAKE?

We see the first question as a formal clarification of the often raised question of how to fix session identifiers and party identifiers in practice. We see the second question as a way to formalize several choices for secure deployment that PAKE standards can offer to end-users: starting from a bare protocol taking only the password, but explaining how to cater to applications that need to fix session identifiers and/or party identifiers externally and bind them to the agreed key. We provide two compilers that transform a PAKE into a bare PAKE, and vice versa. The intuition of both compilers is given in Figure 4. The detailed protocol description in the UC terminology as well as their formal security proofs can be found in Appendix B. Here we only state the main result in form of the following corollary, which follows directly from our Theorems 4 and 5.

Corollary 1. *There exists a non-interactive protocol that tightly realizes $\mathcal{F}_{\mathsf{PAKE}}$ in the $\mathcal{F}_{\mathsf{bPAKE}}$-hybrid model, and there exists a 1-round protocol that tightly realizes $\mathcal{F}_{\mathsf{bPAKE}}$ in the $\mathcal{F}_{\mathsf{PAKE}}$-hybrid model.*



Party $\mathcal{P}$
Party $\mathcal{P}'$
Upon input $\mathcal{P}', ssid, pw$
$\bar{pw} \leftarrow \{\mathcal{P}, \mathcal{P}'\}_{\mathsf{ord}} || ssid | pw$
Upon input $\mathcal{P}, ssid, pw'$
$\bar{pw}' \leftarrow \{\mathcal{P}', \mathcal{P}\}_{\mathsf{ord}} || ssid | pw'$
pw
$\bar{pw}'$
$\mathcal{F}_{\mathsf{bPAKE}}$
K
K'
Output $(ssid, K)$
Output $(ssid, K')$
Party $\mathcal{P}$
Party $\mathcal{P}'$
Upon input pw, A
Sample $N \leftarrow_{\mathrm{R}} \{0,1\}^\kappa$
Upon input pw', B
Sample $N' \leftarrow_{\mathrm{R}} \{0,1\}^\kappa$
N, A
N', B
$ssid \leftarrow \{N, N'\}_{\mathsf{ord}}$
$\bar{pw} \leftarrow \{A, B\}_{\mathsf{ord}} || pw$
$ssid' \leftarrow \{N', N\}_{\mathsf{ord}}$
$\bar{pw}' \leftarrow \{B, A\}_{\mathsf{ord}} || pw'$
$N', ssid, \bar{pw}$
$N, ssid', \bar{pw}'$
$\mathcal{F}_{\mathsf{PAKE}}$
K
K'
Output $(ssid, K)$
Output $(ssid, K')$

Fig. 4. Left: BarePAKE-to-PAKE compiler. Parties use the bare PAKE with their password, appended by all identifiers, without the optional name input. **Right:** PAKE-to-barePAKE compiler. Parties exchange nonces which serve both as their unique party identifier as well as subsession identifier for the PAKE. Their clear-text transmitted names are appended to the password, to ensure authenticity.

5 Password-Only Encrypted Key Exchange

In this section we present a black-box construction of a bare PAKE from a non-interactive key exchange (NIKE) protocol. This modular construction shows that the standard notion of security for NIKE implies, with small computational and bandwidth overhead, a reusable bare PAKE. The construction can be instantiated with a post-quantum NIKE, such as SWOOSH [GdKQ$^+$23], which opens the way for new PAKE constructions based on lattice-based components and possibly post-quantum-secure PAKEs.[17] Furthermore, plugging in the resulting

[17] We do not carry out a security analysis against quantum adversaries because it is not well understood, to the best of our knowledge, on how to deal with quantum adversaries in the ideal cipher model. Nevertheless, we note that the ideal cipher is only

bare PAKE protocol into the bPAKE-to-PAKE transformation allows us to recover some well known results for some Diffie-Hellman-based PAKE protocols that have appeared in the literature.

5.1 Simplified NIKE

We start from the notion of a simplified NIKE (sNIKE), as introduced in [FHKP12]. In contrast to the standard notion of NIKE, where party identities are an input to the protocol, simplified NIKE has only public keys. This notion is well suited to our goal of constructing PAKE protocols that are agnostic of party identities.

Definition 5. *An* sNIKE *scheme is a collection of three efficient algorithms* sNIKE.Setup*,* sNIKE.Keygen*, and* sNIKE.ShKey*, together with a shared key space* $\mathcal{SK}$*.*[18]

- sNIKE.Setup*: On input the security parameter, this randomized algorithm outputs system parameters* prm*.*
- sNIKE.Keygen*: On input* prm*, this randomized algorithm outputs a key pair* (sk, pk)*.*
- sNIKE.ShKey*: On input a secret key* sk *and a public key* pk*, this algorithm outputs either a shared key in* $\mathcal{SK}$ *for the two keys, or a failure symbol* $\perp$*. This algorithm is assumed to always output* $\perp$ *if* sk *is the secret key corresponding to* pk*.*

We say an sNIKE *is* δ-correct *if*

$$\Pr\left[\begin{array}{r}\mathsf{sNIKE.ShKey}(sk_1, pk_2) = \\ \mathsf{sNIKE.ShKey}(sk_2, pk_1)\end{array}\;\middle|\;\begin{array}{l}prm \leftarrow\!\!\$\ \mathsf{sNIKE.Setup}(1^\kappa) \\ (sk_1, pk_1) \leftarrow\!\!\$\ \mathsf{sNIKE.Keygen}(prm) \\ (sk_2, pk_2) \leftarrow\!\!\$\ \mathsf{sNIKE.Keygen}(prm)\end{array}\right] \geq 1 - \delta(\kappa)\,.$$

An sNIKE *scheme is* perfectly correct *if this holds for* $\delta(\kappa) = 0$*.*

Security. Later in this section we will give a new generic construction of bare PAKE from sNIKE and show that its security follows from the vanilla security notion for sNIKE, which we recall here. Figure 5 shows the security game for sNIKE. The notion was originally introduced by Cash, Kiltz and Shoup [CKS08] and hence we call it (CKS). The same figure shows, in blue, additional restrictions that define *one-time CKS* security (OT-CKS). This leads to a weaker version of

relevant when protecting against active attacks, which means that passive security (even with a posteriori password compromise) follows directly from the security of the underlying NIKE. This means that our protocol is suitable for applications that are concerned with preserving the confidentiality of data exchanged in the presence of passive attackers today, which may log the data and have access to a quantum computer in the future.

[18] In practice, $\mathcal{SK}$ will be the set of bit strings of fixed length ℓ for some $\ell \geq \kappa$. We consider the more general case to cover core protocols such as Diffie-Hellman, where keys are elements of a group with order at least 2^κ.

```
Game $\mathsf{CKS}^{\mathsf{sNIKE}}_{\mathcal{A}}(1^\kappa, b)$
$\mathsf{hoks}, \mathsf{coks} \leftarrow \{\,\}$
$\mathsf{shk}[x] \leftarrow \bot$ for all $x$
$\mathsf{otused} \leftarrow \{\,\}$
$prm \leftarrow_{\mathrm{R}} \mathsf{sNIKE.Setup}(1^\kappa)$
$b' \leftarrow_{\mathrm{R}} \mathcal{A}^{\mathcal{O}}(prm)$
Return $b'$

Oracle $\mathsf{RegHonest}()$
$(sk, pk) \leftarrow_{\mathrm{R}} \mathsf{sNIKE.Keygen}(prm)$
$\mathsf{hoks}[pk] \leftarrow sk$
Return $pk$

Oracle $\mathsf{RegCorrupt}(pk)$
If $pk \notin \mathsf{hoks} \cup \mathsf{coks}$
    $\mathsf{coks} \leftarrow \mathsf{coks} \cup \{pk\}$
```

```
Oracle $\mathsf{CorrReveal}(pk_1, pk_2)$
If $pk_1 \notin \mathsf{hoks} \vee pk_2 \notin \mathsf{coks}$ Return $\bot$
If $pk_1 \in \mathsf{otused}$ Return $\bot$
$\mathsf{otused} \leftarrow \mathsf{otused} \cup \{pk_1\}$
$sk_1 \leftarrow \mathsf{hoks}[pk_1]$
Return $\mathsf{sNIKE.ShKey}(sk_1, pk_2)$

Oracle $\mathsf{Test}(pk_1, pk_2)$
If $pk_1 \notin \mathsf{hoks} \vee pk_2 \notin \mathsf{hoks}$ Return $\bot$
If $pk_1 \in \mathsf{otused}$ Return $\bot$
$\mathsf{otused} \leftarrow \mathsf{otused} \cup \{pk_1\}$
$sk_1 \leftarrow \mathsf{hoks}[pk_1]$
$K_0 \leftarrow \mathsf{sNIKE.ShKey}(sk_1, pk_2)$
If $\{pk_1, pk_2\}_{\mathsf{ord}} \notin \mathsf{shk}$
    $\mathsf{shk}[\{pk_1, pk_2\}_{\mathsf{ord}}] \leftarrow_{\mathrm{R}} \mathcal{SK}$
$K_1 \leftarrow \mathsf{shk}[\{pk_1, pk_2\}_{\mathsf{ord}}]$
Return $K_b$
```

Fig. 5. CKS-style security game for an sNIKE scheme [CKS08, FHKP12]. Oracle $\mathcal{O}$ provides adversary $\mathcal{A}$ with access to oracles $\mathsf{RegHonest}, \mathsf{RegCorrupt}(\cdot), \mathsf{CorrReveal}(\cdot,\cdot), \mathsf{Test}(\cdot,\cdot)$. Variables hoks and coks represent the sets of respectively honest and corrupt keys, and shk is a table of shared keys assigned to a pair of honest keys. Blue code enforces one-time-use restrictions on secret keys, and it corresponds to a weaker notion of CKS security which we call OT-CKS.

sNIKE security, which we will show is sufficient to construct one-time (non-reusable) bare PAKE, and hence also PAKE. We will further show that this weaker version of one-time sNIKE security can be achieved in the random oracle model starting from a passive one-way secure sNIKE, of which a prominent case is the textbook Diffie-Hellman protocol under the CDH assumption. A corollary of these results is a modular restating of previous results showing that UC PAKE can be constructed directly from CDH in the joint ideal-cipher and random oracle model.

Remark 1. Any sNIKE that is not correct with overwhelming probability cannot satisfy either security notion: an attacker will observe an inconsistency of keys if $b = 0$ that is not present when $b = 1$.

Remark 2. Also note that we do not deal with *reflexive* attacks by assuming (see above note on the sNIKE.ShKey algorithm) that a party checks that it is not deriving a key with itself and outputs $\bot$ on such key derivation queries. In general, proving security when these checks are not made (e.g., for performance reasons) requires additional computational assumptions. However, the theorems in this section can be extended to this case to show that EKE transforms a secure sNIKE that omits those checks and withstands reflexive attacks into a UC-secure bare PAKE protocol.

Definition 6. *We say that* sNIKE *is* (m,n)-CKS*, or* CKS-secure *for* m *honest keys and* n *corrupt keys if, for every ppt adversary* $\mathcal{A}$ *placing at most* m *queries to* $\mathsf{RegHonest}$ *and* n *queries to* $\mathsf{RegCorrupt}$*, the following advantage function is negligible in the security parameter* κ*.*

$$\mathbf{Adv}^{(m,n)\text{-}\mathsf{CKS}}_{\mathcal{A},\mathsf{sNIKE}}(\kappa) := |\Pr[\mathsf{CKS}^{\mathsf{sNIKE}}_{\mathcal{A}}(1^\kappa,1) \Rightarrow 1] - \Pr[\mathsf{CKS}^{\mathsf{sNIKE}}_{\mathcal{A}}(1^\kappa,0) \Rightarrow 1]| \,.$$

We say that sNIKE *is* (m,n)-OT-CKS*, or* one-time CKS-secure*, for* m,n *as above, if this holds when the one-time restrictions are in place.*

We also define a minimal security notion for sNIKE, namely a *one-way* (OW) counterpart of $(2,0)$-CKS, where there are only two honest keys, there are no corrupt keys, and the adversary must explicitly compute the session key agreed between the two honest keys. We specify this security experiment in Figure 6. Note that in this experiment a secret key is used at most once, and so the one-time restriction is redundant.

```
Game $\mathsf{OW}^{\mathsf{sNIKE}}_{\mathcal{A}}(1^\kappa)$
$prm \leftarrow_{\mathrm{R}} \mathsf{sNIKE.Setup}(1^\kappa)$
$(sk_1, pk_1) \leftarrow_{\mathrm{R}} \mathsf{sNIKE.Keygen}(prm)$
$(sk_2, pk_2) \leftarrow_{\mathrm{R}} \mathsf{sNIKE.Keygen}(prm)$
$(i, j, K') \leftarrow_{\mathrm{R}} \mathcal{A}(prm, pk_1, pk_2)$
$K \leftarrow \mathsf{sNIKE.ShKey}(sk_i, pk_j)$
Return 1 if $K' = K \neq \bot$ and 0 otherwise
```

Fig. 6. One-wayness game for an sNIKE scheme.

Definition 7. *We say that* sNIKE *is* OW*, or* one-way secure*, if for every ppt adversary* $\mathcal{A}$*, the following advantage function is negligible in the security parameter* κ*.*

$$\mathbf{Adv}^{\mathsf{OW}}_{\mathcal{A},\mathsf{sNIKE}}(\kappa) := \Pr[\mathsf{OW}^{\mathsf{sNIKE}}_{\mathcal{A}}(1^\kappa) \Rightarrow 1] \,.$$

We now show that the above one-wayness notion implies general one-time secure sNIKE in the Random Oracle Model. For any sNIKE scheme let $\mathsf{H}[\mathsf{sNIKE}]$ be the same scheme except the shared key output is "post-processed" using a hash function H, i.e., $\mathsf{H}[\mathsf{sNIKE}].\mathsf{ShKey}(sk,pk) = \mathsf{H}(\mathsf{sNIKE.ShKey}(sk,pk))$. In what follows, we will call $\mathsf{sNIKE.ShKey}(sk,pk)$ a *pre-key*. The proof of the following theorem is given in Appendix C.1. It is a direct reduction, where we leverage the fact that one can correctly program the Random Oracle to justify a single corrupt reveal query by guessing which point in the Random Oracle table could allow the adversary to detect an inconsistent simulation.

Theorem 1 ($\mathsf{OW} \Rightarrow \mathsf{OT}$-$\mathsf{CKS}$)**.** *Let* sNIKE *be a* δ*-correct and* OW*-secure simplified NIKE with shared key space* $\mathcal{SK}$*. Then, if we model hash function* $\mathsf{H} : \mathcal{SK} \rightarrow$

$\{0,1\}^\kappa$ *as a random oracle, for every attacker* $\mathcal{A}$ *against the* (m,n)-$\mathsf{OT\text{-}CKS}$ *security of* $\mathsf{H}[\mathsf{sNIKE}]$*, making at most* q_H *queries to* H *and placing at most* q_T *queries to* Test*, there exists an attacker* $\mathcal{B}$ *in* OW *security game of* sNIKE *such that*

$$\mathbf{Adv}^{(m,n)\text{-}\mathsf{OT\text{-}CKS}}_{\mathcal{A},\mathsf{H}[\mathsf{sNIKE}]}(\kappa) \leq q_T \cdot \delta(\kappa) + m^2 \cdot (q_H+1)^2 \cdot \mathbf{Adv}^{\mathsf{OW}}_{\mathcal{B},\mathsf{sNIKE}}(\kappa)\,,$$

One-time NIKE from CDH. We now consider the classical construction of a NIKE known as hashed Diffie-Hellman (HDH):

- $\mathsf{sNIKE.Setup}$: On input the security parameter, outputs $prm = (\mathbb{G}, g, q, H)$, the description of a cyclic group $\mathbb{G}$ of prime order q and generator g, along with a hash function mapping $\mathbb{G}$ to $\{0,1\}^\kappa$.
- $\mathsf{sNIKE.Keygen}$: On input prm, this randomized algorithm outputs a key pair (g^a, a), where a is generated as $a \leftarrow_{\mathrm{R}} \mathbb{Z}_q$.
- $\mathsf{sNIKE.ShKey}$: On input a public key pk_1 and a secret key $sk_2 = a$, this algorithm outputs $H(pk_1^a)$ if $pk_1 \neq g^a$ and $\perp$ otherwise.

Corollary 2. *Hashed Diffie-Hellman is a* (m,n)-$\mathsf{OT\text{-}CKS}$ *secure sNIKE under the CDH assumption, when we model* H *as a random oracle. More precisely, for every one-time* sNIKE *attacker* $\mathcal{A}$ *against HDH making at most* q_H *queries to* H *and* q_T *queries to the Test oracle, there exists an algorithm* $\mathcal{B}$ *running in essentially the same time as* $\mathcal{A}$ *such that*

$$\mathbf{Adv}^{\mathsf{OT\text{-}CKS}}_{\mathcal{A},\mathsf{HDH}}(\kappa) \leq m^2 \cdot (q_H+1)^2 \cdot \mathbf{Adv}^{\mathsf{CDH}}_{\mathcal{B},\mathbb{G}}(\kappa).$$

Proof. The corollary follows by observing that the classic unauthenticated Diffie-Hellman protocol a perfectly correct sNIKE and its one-way security is exactly the CDH problem. □

Remark 3. We present this corollary here because, combined with the results that will follow in the rest of this section, it provides a clear and modular explanation of why one can prove that EKE-HDH is a UC-secure PAKE assuming only CDH (in the combined Ideal Cipher and Random Oracle Model). It also opens the way for constructing PAKE from one-way secure lattice-based NIKE, which may eliminate the need for costly NIZK computations and bandwidth.

Remark 4. In the particular case of HDH, it is well known that much tighter reductions can be obtained using so-called strong DH assumptions. In particular, HDH is a tightly CKS-secure sNIKE assuming Gap DH: a direct reduction can be constructed that generates all honest key pairs by randomizing the Gap DH challenge, and answers corrupt-reveal queries using the gap oracle. Note, however, that the reduction in this case makes significantly more work than running the adversary and $O(q_H^2)$ queries to the gap oracle. This overhead in the reduction can be reduced by including the public keys in the input to the key derivation hash, in which case the number of gap oracle queries is linear in q_H.

5.2 The EKE Construction

Figure 7 shows the canonical “non-interactive” (SSF) bare PAKE using the EKE blueprint, generalized from Diffie-Hellman to any sNIKE. The protocol is defined in the $\mathcal{F}_{crs}$-hybrid model, where the common-reference-string distribution is defined by the sNIKE global parameter generation algorithm $\mathsf{sNIKE.Setup}$.[19] Throughout the discussion, for conciseness, we will keep $\mathcal{F}_{crs}$ implicit and assume that all parties have access to prm.

```
algorithm StartIni($prm, pw, id$):
    $(sk, pk) \leftarrow_{\mathrm{R}} \mathsf{sNIKE.Keygen}(prm)$
    $c \leftarrow \mathsf{IC}_{pw}(pk)$
    $m \leftarrow (c, id)$
    $st \leftarrow (sk, pw, m)$
    Return $(m, st)$
```

```
algorithm EndIni($m', st$):
    Parse $(sk, pw, m) \leftarrow st$
    Parse $(c', id') \leftarrow m'$
    $pk' \leftarrow \mathsf{IC}_{pw}^{-1}(c')$
    $\bar{K} \leftarrow \mathsf{sNIKE.ShKey}(pk', sk)$
    $ssid \leftarrow \{m, m'\}_{\mathsf{ord}}$
    $K \leftarrow \mathsf{KDF}(\bar{K}, ssid)$
    Return $(ssid, id', K)$
```

Fig. 7. Protocol EKE-NIKE: A bare password-encrypted key exchange (EKE) based on sNIKE and an ideal cipher IC over the domain of NIKE public keys. The protocol is single simultaneous flow (SSF). The setup algorithm run by $\mathcal{F}_{crs}$ is Setup = $\mathsf{sNIKE.Setup}$.

Theorem 2 (Security of EKE-NIKE). *Let π be the bPAKE protocol that results from combining EKE-NIKE in Figure 7 with wrapper code π_{SH} (cf. Section 3.3). Then π UC-emulates $\mathcal{F}_{\mathsf{bPAKE}}$ under static corruptions, assuming* IC *is an ideal cipher on domain $\mathcal{C} = \mathcal{PK}$,* sNIKE *is CKS-secure, and the distribution of public keys produced by* $\mathsf{sNIKE.Keygen}$ *is computationally close to uniform over $\mathcal{PK}$ and has min-entropy at least κ bits.*

More precisely, the UC emulation bound is given by

$$\mathcal{D}_{\mathcal{Z}}^{\pi,\{\mathcal{F}_{\mathsf{bPAKE}},\mathsf{Sim}\}}(\kappa) \leq \mathbf{Adv}_{\mathcal{B},\mathsf{sNIKE}}^{(q_{\mathsf{IC}},q_{\mathsf{IC}})\text{-}\mathsf{CKS}}(\kappa) + q_K \cdot \epsilon_{\mathsf{KDF}} + q_{\mathsf{IC}} \cdot \epsilon_{\mathsf{sNIKE.Keygen}} + 2 \cdot q_{\mathsf{IC}}^2 \cdot (1/|\mathcal{PK}| + 1/2^{\kappa})$$

where $\mathcal{D}_{\mathcal{Z}}^{\pi,\{\mathcal{F}_{\mathsf{bPAKE}},\mathsf{Sim}\}}(\kappa)$ is the distinguishing advantage of environment $\mathcal{Z}$ between the real world execution of π and the simulation presented by Sim *interacting with $\mathcal{F}_{\mathsf{bPAKE}}$, q_{IC} is an upper bound on the total number of IC computations, q_K is an upper bound on the number of session keys derived by honest parties, ϵ_{KDF} is the maximum distinguishing advantage of ppt adversaries against the PRF property of KDF, and $\epsilon_{\mathsf{sNIKE.Keygen}}$ is the maximum distinguishing advantage of ppt adversaries in distinguishing public-keys produced by* $\mathsf{sNIKE.Keygen}$ *from uniform values in $\mathcal{PK}$.*

[19] The $\mathcal{F}_{crs}$-hybrid model assumes that all parties have access to a functionality that publishes a common reference string (CRS) sampled from some distribution. In our work this CRS does not need to be programmed by the simulator, and hence this functionality can be global.

```
Messages from $\mathcal{F}_{\mathsf{bPAKE}}$:

On ($\mathsf{NewSession}, sid, \mathcal{P}, i, id, \mathsf{role}$):
  – Ignore if record $\langle \mathcal{P}, i, * \rangle$ exists or if $\mathsf{role} \neq \perp$
  – $c \leftarrow \mathsf{IC}_{\mathcal{S}}()$
  – $m \leftarrow (c, id)$
  – $st \leftarrow (\perp, \perp, m)$
  – Store $\langle \mathcal{P}, i, st \rangle$
  – Send $(i, m)$ to $\mathcal{A}$

Messages from $\mathcal{A}$:

On message $(sid, i, m')$ from $\mathcal{A}$ towards honest $\mathcal{P}$:
  – Ignore if record $\langle \mathcal{P}, i, [st] \rangle$ does not exist
  – Parse $(\perp, \perp, m) \leftarrow st$
  – Parse $(c, id) \leftarrow m$
  – Parse $(c', id') \leftarrow m'$
  – $ssid \leftarrow \{m, m'\}_{\mathsf{ord}}$
  – If record $\langle [\mathcal{P}'], [i'], (\cdot, \cdot, m') \rangle$ exists:
      • Call ($\mathsf{PassiveNewKey}, sid, \mathcal{P}, i, \mathcal{P}', i', ssid$)
  – If record $\langle [\mathcal{P}'], [i'], (\cdot, \cdot, (c', \cdot)) \rangle$ exists:
      • Call ($\mathsf{ActiveNewKey}, sid, \mathcal{P}, i, \perp, \perp, ssid, id'$)
  – If $L[c'] = \perp$: // covers $(\cdot, \cdot, \cdot, c') \notin T$
      • Call ($\mathsf{ActiveNewKey}, sid, \mathcal{P}, i, \perp, \perp, ssid, id'$)
  – $(pw, pk') \leftarrow L[c']$
  – $(sk', pk) \leftarrow \mathsf{IC}_{\mathcal{S}}^{-1}(pw, c)$
    // $sk' \neq \perp$ as $c$ was generated by simulator
  – $\bar{K} \leftarrow \mathsf{sNIKE.ShKey}(sk', pk')$
  – $K \leftarrow \mathsf{KDF}(\bar{K}, ssid)$
  – Call ($\mathsf{ActiveNewKey}, sid, \mathcal{P}, i, pw, K, ssid, id'$)

Ideal cipher calls:

On $\mathsf{IC}_{\mathcal{S}}()$:
  – $c \leftarrow_{\mathrm{R}} \{c' \in \mathcal{C} \mid (\cdot, \cdot, \cdot, c') \notin T\}$
  – Append $(\perp, \perp, \perp, c)$ to $T$
  – Return $c$

On $\mathsf{IC}_{\mathcal{S}}^{-1}()$:
  – If $(pw, [pk], [sk], c) \in T$, return $(pk, sk)$
    // may return $sk = \perp$
  – $(sk, pk) \leftarrow_{\mathrm{R}} \mathsf{sNIKE.Keygen}(prm)$
  – Abort if $(pw, pk, \cdot, \cdot) \in T$
  – Append $(pw, pk, sk, c)$ to $T$
  – Return $(sk, pk)$

On $\mathsf{IC}_{pw}(pk)$:
  – If $(pw, pk, \cdot, [c]) \in T$, return $c$
  – $c \leftarrow_{\mathrm{R}} \{c' \in \mathcal{C} \mid (\cdot, \cdot, \cdot, c') \notin T\}$
  – Append $(pw, pk, \perp, c)$ to $T$
  – $L[c] \leftarrow (pw, pk)$
  – Return $c$

On $\mathsf{IC}_{pw}^{-1}(c)$:
  – $(sk, pk) \leftarrow \mathsf{IC}_{\mathcal{S}}^{-1}(pw, c)$
  – Return $pk$
```

Fig. 8. Simulator for the proof of Theorem 2. The simulator runs $\mathsf{sNIKE.Setup}$ on start-up and provides the resulting prm to the adversary as the output of $\mathcal{F}_{crs}$. These are also used throughout the simulation.

Proof. (Sketch.) The full proof is given in Appendix C.2. Here we give only a sketch. We present the simulator that justifies our protocol in Figure 8. The simulator generates random IC ciphertexts as the outgoing messages of honest parties, so as not to commit to an input key-pair (it does not know under which password to encrypt it). This is undetectable to the attacker unless it guesses one of these public keys, which we exclude using the assumption that they are high-entropy. The inverse operation of the ideal cipher is simulated by generating key pairs, whenever the adversary tries to decrypt a fresh ciphertext—in particular the ones that honest parties produced—hence the assumption that public keys look uniform. This will allow the simulator to a-posteriori recover the secret key of an honest party, when it extracts a correct password guess from the adversary (see below).

When an adversary delivers an encrypted key to an honest party, we have two options: either 1) the ciphertext was honestly generated, or 2) it was maliciously generated. If 1) occurred, then the adversary is launching a passive attack (this may be only partially passive if the adversary alters the rest of the message) and will not know the associated session key down to the security of the underlying sNIKE scheme; in this case the simulator calls the functionality on $\mathsf{PassiveNewKey}$ or $\mathsf{ActiveNewKey}$ without a password guess, depending on whether the attack is passive, or partially passive. If 2) occurred, then the simulator still needs to deal with two subcases: i) it can extract the password associated with that ciphertext; or ii) it cannot extract the password. In the first subcase the simulator constructs a plausible protocol execution for a possible correct password guess (as described above) and calls $\mathsf{ActiveNewKey}$. In the second subcase we show that the adversary simply has no control of the public key that the honest party would recover from that ciphertext. In all cases where the functionality chooses a random session key, we show that the attacker cannot distinguish this from the real-world down to sNIKE security. In the cases that the password is extracted and it is correct, the simulator perfectly mimics the behavior of the honest party. □

Remark 5. Suppose EKE-sNIKE is used as a PAKEas proposed in Figure 14, i.e., each instance of the protocol is used to process at most one incoming message. Then, the proof above still works, but the reduction to sNIKE security is now a valid reduction to *one-time* sNIKE security. Combined with the results on HDH sNIKE given in the beginning of this section, this gives us an alternative proof that EKE-HDH is a UC-secure PAKE down to CDH. Furthermore, it also allows us to plug-in a passively secure lattice-based sNIKE such as the core of SWOOSH [GdKQ$^+$23], which may open the way for post-quantum secure PAKE from lattices without relying on costly NIZK components.

We note that our proof applies when there exists domain such that 1) the distribution (of some encoding) of NIKE public keys is indistinguishable from uniform over said domain, and 2) there is an ideal cipher over domain said domain. Therefore, if the NIKE public keys can be encoded as uniform elements of some domain, the question remains how to instantiate the IC over this domain in practice. The half-ideal cipher abstraction, which is an IC domain extender for some algebraic domains, is one possible solution for instantiation but requires indifferentiable hashing to the public-key space. It is an open question whether isogeny-based NIKEs satisfy these properties.

Remark 6. These results (see also the discussion on adaptive corruptions in Section 7) give strong evidence that reusable bPAKE is harder to construct than standard PAKE. This does not stand in contention with the results we gave in Section 4 with respect to being able to construct PAKE from bare PAKE and vice-versa. Indeed, although the construction of PAKE from bare PAKE is really showing an implication, the construction of bare PAKE from PAKE is running many independent instances of the PAKE protocol for the same password, rather than reusing components previously computed by the party for the same password.

Tightness of Theorem 2. Note that the reduction of NIKE security to to bPAKE is tight. Nevertheless, the NIKE advantage may depend on q_{IC}, which may be of the order of 2^{κ}, and this maps to the number of NIKE public keys. This means that the tightness of EKE-NIKE depends crucially on the security bound for the underlying NIKE. In particular, if the NIKE bound does not depend on this number, then neither does the EKE-NIKE bound. This is the case, for example, for HDH NIKE that has a tight reduction to gap-CDH. Moreover, if EKE-NIKE is used as a non-reusable bPAKE, security requires only a *one-time* NIKE, which allows for NIKE instantiations with tighter reductions and/or weaker assumptions.

6 Password-Only CPace

In Figure 9 we cast the CPace PAKE protocol [HL17, AHH21] as a bare PAKE. The differences to "basic CPace" (referred to as "CPace" in the below) of Abdalla et al. [AHH21] are as follows.

- Bare CPace derives the **group generator only from** pw, while CPace derives the generator from pw and both party identifiers. If multiple keys need to be exchanged, or more than two parties use the protocol, CPace also needs to additionally make the generator computation dependent on the session identifier sid [AHH21].
- Bare CPace lets parties **send their own name** alongside their DH message, while in CPace parties retrieved the counterparty name from the application (i.e., as input).
- CPace computes the session key as a hash of the Diffie Hellman value and the DH public keys. Bare CPace requires to additionally **include the party names into the final key derivation hash**, to achieve the desired authentication properties of the party names (matching output keys imply that parties reliably received the name of their respective counterparty).
- Bare CPace lets a party **output the counterparty's name**, while CPace required it as input to derive the password as described above.
- Bare CPace lets parties **output a subsession (key exchange) identifier** $ssid$ which is composed of the two party names and messages, and which is unique with overwhelming probability if output by an honest party. Basic CPace required such a unique session identifier as input.

We now specify the algorithms for the body of bare CPace: $\mathrm{Setup}_{\mathsf{bCPace}}$, $\mathrm{StartIni}_{\mathsf{bCPace}}$, $\mathrm{EndIni}_{\mathsf{bCPace}}$. Because CPace is a single-simultaneous flow protocol, we can omit the receiver algorithms StartRsp and EndRsp. The protocol is defined in the $\mathcal{F}_{crs}$-hybrid model, where the common-reference-string distribution is defined by the parameter generation algorithm $\mathrm{Setup}_{\mathsf{bCPace}}$.

- $\mathrm{Setup}_{\mathsf{bCPace}}(1^\kappa)$ takes as input the security parameter and produces public parameters prm, containing a group $\mathbb{G}$ of order q and hash functions $H : \{0,1\}^* \to \{0,1\}^\kappa$, $\mathcal{H}_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$.
- $\mathrm{StartIni}_{\mathsf{bCPace}}(prm, pw, id)$ takes as input prm, a password pw and a party name id. It then computes $g \leftarrow \mathcal{H}_{\mathbb{G}}(pw)$, samples $a \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, sets $A \leftarrow g^a$ and outputs $m_1 \leftarrow (A, id)$ and $st_I \leftarrow (pw, id, g, a, A)$.

- $\mathrm{EndIni}_{\mathsf{bCPace}}(st_I, m_2)$ takes as input a state $st_I := (pw, id, g, a, A)$ and a message $m_2 := (B, cpid)$. It then sets $ssid \leftarrow \{(A||id), (B||cpid)\}_{\mathsf{ord}}$, computes $K \leftarrow H(B^a, \{A, B\}_{\mathsf{ord}})$ and outputs K, $ssid$, and $cpid$.

We denote with Π_{bCPace} the structured protocol with the shell executing the wrapper code π_{SH} (cf. Section 3.3), calling the body's algorithms $\mathrm{Setup}_{\mathsf{bCPace}}$, $\mathrm{StartIni}_{\mathsf{bCPace}}$, $\mathrm{EndIni}_{\mathsf{bCPace}}$ specified above. For clarity, we state Π_{bCPace} below, marking in gray the protocol parts that run in the body. Because CPace is single-simultaneous flow and hence does not have initiator and responder roles, we assume wlog that parties retrieve inputs with $\mathsf{role} = \bot$.

Structured protocol Π_{bCPace}

```
On creation, do:
  ∘ generate group $\mathbb{G}$ of order $q$  //$\mathrm{Setup}_{\mathsf{bCPace}}$
  ∘ pick functions $H : \{0,1\}^* \to \{0,1\}^\kappa$, $\mathcal{H}_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$  //$\mathrm{Setup}_{\mathsf{bCPace}}$
  ∘ store $prm \leftarrow (\mathbb{G}, q, H, \mathcal{H}_{\mathbb{G}})$

On ($\mathsf{NewSession}$, $sid, i, pw, id, \bot$), do:
  ∘ ignore this query if record $\langle [i], ... \rangle$ exists
  ∘ $g \leftarrow \mathcal{H}_{\mathbb{G}}(pw)$, sample $a \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, set $A \leftarrow g^a$  //$\mathrm{StartIni}_{\mathsf{bCPace}}(prm, pw, id)$
  ∘ set $st \leftarrow (pw, id, g, a, A)$
  ∘ store $\langle i, st \rangle$
  ∘ send message $(sid, i, (A, id))$ to $\mathcal{A}$

When $\mathcal{A}$ delivers message $(sid, i, (B, cpid))$, do:
  ∘ retrieve record $\langle [i], (pw, id, g, a, A) \rangle$, ignore message otherwise
  ∘ $ssid \leftarrow \{(A||id), (B||cpid)\}_{\mathsf{ord}}$, $K \leftarrow H(B^a, ssid)$  //$\mathrm{EndIni}_{\mathsf{bCPace}}(st, (B, cpid))$
  ∘ output $(sid, i, K, ssid, cpid)$
```

Fig. 9. Bare CPace as structured protocol Π_{bCPace}. Gray parts are run in body, the rest is the wrapper code π_{SH} executed in the shell.

Theorem 3 (Security of bare CPace). *Protocol Π_{bCPace} UC-emulates the* lazy-extraction *version of the bare PAKE functionality, $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$, shown in Figure 2, in the $\mathcal{F}_{crs}$-hybrid model, with $\mathcal{H}_{\mathbb{G}}, H$ modeled as random oracles, if the gapCDH and sim-gapCDH assumptions hold in $\mathbb{G}$, and with respect to static party corruption.*

That is, for any efficient adversary $\mathcal{A}$ against Π_{bCPace} there exists an efficient simulator Sim *that interacts with $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$ and produces a view such that for all efficient environments it holds that*

$$\mathcal{D}_{\mathcal{Z}}^{\Pi_{\mathsf{bCPace}}, \{\mathcal{F}_{\mathsf{bPAKE^{LE}}}, \mathsf{Sim}\}}(\kappa) \leq \frac{(q_{\mathsf{H2G}}+q_{\mathsf{var}})^2+q_{\mathsf{ns}}^2}{2 \cdot 2^\kappa} + q_{\mathsf{var}} q_{\mathsf{pw}} \mathbf{Adv}_{\mathbb{G}}^{\mathsf{gapCDH}} + q_{\mathsf{H2G}}^2 q_{\mathsf{pw}} \mathbf{Adv}_{\mathbb{G}}^{\mathsf{sim-gapCDH}}$$

where $\mathcal{D}_{\mathcal{Z}}^{\Pi_{\mathsf{bCPace}},\{\mathcal{F}_{\mathsf{bPAKE^{LE}}},\mathsf{Sim}\}}(\kappa)$ *is the distinguishing advantage of environment* $\mathcal{Z}$ *between the real world execution of* Π_{bCPace} *and the simulation presented by* Sim *interacting with* $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$*, and where* q_{var} *is the overall number of passwords in the system,* q_{H2G} *is the number of* $\mathcal{H}_{\mathbb{G}}$ *queries issued by* $\mathcal{A}$*,* q_{ns} *is the number of* $\mathsf{NewSession}$ *queries issued by* $\mathcal{A}$*, and* q_{pw} *is the maximum number of parties receiving the same password through a* $\mathsf{NewSession}$ *input.*

Proof Sketch. The high level idea of the proof is as follows. Starting from the real execution where environment $\mathcal{Z}$ interacts with parties running Π_{bCPace} and a dummy adversary $\mathcal{A}$, we subsequently change the execution until we end up with $\mathcal{Z}$ interacting with functionality $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$ and a simulator Sim. The two main challenges are as follows: the simulator needs to produce message indistinguishable from real ones without knowing the passwords of honest parties, and we have to randomize the output keys of parties. The simulation approach is to use a common "simulation" generator g_{Sim} to generate all group elements. The simulator maintains trapdoors $r \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ from queries $\mathcal{H}_{\mathbb{G}}(pw) = g_{\mathsf{Sim}}^r$ for adversarial password hashes, and uses these trapdoors to identify password guesses. In a bit more detail, output keys in CPace are computed from hashing Diffie-Hellman keys, e.g., $H(K, ssid)$, where $K := \mathcal{H}_{\mathbb{G}}(pw)^{ab}$ and a and b are the secret keys of Alice and Bob sharing the key. Since Sim does not know $\mathcal{H}_{\mathbb{G}}(pw)$ when simulating Alice's message $A := g_{\mathsf{Sim}}^s$, Sim does not explicitly know Alice's secret key a. Hence Sim cannot compute $K = B^a$ from Bob's message B. However, *after* learning Alice's password pw, Sim's knowledge of trapdoor r with $\mathcal{H}_{\mathbb{G}}(pw) = g_{\mathsf{Sim}}^r$ lets Sim compute Alices "correct" secret key as s/r, since $A = g_{\mathsf{Sim}}^s = (g_{\mathsf{Sim}}^r)^{s/r} = \mathcal{H}_{\mathbb{G}}(pw)^{s/r}$.

Our proof proceeds as follows. First, we switch the simulation to the common simulation base g_{Sim} and let Sim remember trapdoors for all group elements, i.e., $\mathcal{H}_{\mathbb{G}}$ queries, and simulated messages of honest parties. Then, we randomize the output keys of parties. For honest sessions, keys are pseudorandom under the gapCDH assumption, since keys output by parties are hashed Diffie-Hellman keys (requiring the environment to solve for these DH keys if it aims at detecting the randomization). However, the reduction requires a DDH oracle (hence the security of CPace relies on gap-type assumptions) to consistently simulate other keys output by the parties. For attacked parties, the randomization of keys is more complex, because the simulator has to extract password guesses from adversarial messages reaching the attacked party. Here, we can show that if the sim-gapCDH assumption holds in $\mathbb{G}$, the adversary cannot manufacture adversarial messages that constitute a guess for two different passwords. Sim can then extract the unique guess from the adversarial queries to $\mathcal{H}_{\mathbb{G}}$ and H. After randomizing all output keys, randomizing the protocol transcript can simply leverage the entropy of secret keys, since no other simulated values depend on the password anymore. Because $\mathbb{G}$ is cyclic, $\mathcal{H}_{\mathbb{G}}(pw)$ is a generator and the (whp) uniqueness of secret keys is enough to argue a uniform distribution of honest parties' messages.

With outputs being determined by $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$ and the transcript being sampled at random from the group, the simulation does not depend on the passwords anymore and the ideal execution is reached, concluding the proof. The detailed proof is in Appendix D, and we depict the simulator in Figure 19.

Tightness of Theorem 3. The proof of Theorem 3 established the security of CPace in a *multi-session* setting, where sessions are allowed to be re-used and can output multiple keys. Compared to the standard and *non-reusable* CPace [AHH21], we have an additional cost of a factor q_{pw} for sim-gapCDH and a factor approximately linear in the number of new sessions for gap-CDH. The latter is attributed to the 1-to-n nature of reusable PAKE (i.e., every input password can result in the output of n keys), while standard PAKE is 1-to-1. This also means that a 1-to-1 bare CPace, where a party only outputs one key per password, is almost as tight as CPace with session identifiers [AHH21]: If everybody uses a different password (as in the PAKE built from bare CPace via the bPAKE-to-PAKE transform in Figure 4) then $q_{\mathsf{pw}} = 1$ and the bounds of our proof and the proof of [AHH21] would be exactly equal.

7 Security under Adaptive Corruptions

Adaptive corruptions refer to party corruptions that occur at a point in time where that a party already executed some parts of the protocol honestly. Upon corruption, the whole internal state produced up to this point is revealed to the adversary, and from that point on, the adversary controls all actions of the corrupted party.

Adaptive corruptions are a challenging but realistic attack scenario, and in this section we argue that some of the protocols we consider in this work maintain their guarantees even when adaptive corruptions are allowed. In a nutshell, the challenge of simulating adaptive corruptions lies in manufacturing secret values of the corrupted party that "explain" both its inputs and its sent messages up to the point of corruption. Here, note that the messages were simulated *without* knowledge of the secret inputs, i.e., the passwords.

```
On $(\mathsf{AdaptiveCorruption}, sid, \mathcal{P})$ from $\mathcal{A}$:
  – Initialize an empty array $\mathsf{state}$
  – For each record $\langle \mathcal{P}, [i, pw, id] \rangle$:
      • For each record $\langle \mathsf{sesinf}, \mathcal{P}, i, [ssid, id', k] \rangle$:
          ∗ set $\mathsf{state}[ssid] \leftarrow (i, pw, id, id', k)$
  – Send $\mathsf{state}$ to $\mathcal{A}$
```

Fig. 10. Interface for adaptive corruptions in $\mathcal{F}_{\mathsf{bPAKE}}$ (or variants thereof).

```
On $(\mathsf{AdaptiveCorruption}, sid, \mathcal{P})$ from $\mathcal{A}$:
  – Send $(\mathsf{AdaptiveCorruption}, sid, \mathcal{P})$ to $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$ and receive back $\mathsf{state}$
  – For each $ssid$ in $\mathsf{state}$:
      • Parse $\mathsf{state}[ssid]$ as $(i, pw, id, id', k)$
      • Parse $ssid := \{(A||id), (B||id')\}_{\mathsf{ord}}$, i.e., $(sid, i, (B, id'))$ is the message that led $\mathcal{P}$ to output $k$
      • Retrieve record $(\mathcal{P}, i, [s, S], id, [\mathsf{keys}_i])$
      • Do once for every $i$:
          ∗ If there is a record $(\mathcal{H}_{\mathbb{G}}, pw, [r, R])$ then set $y_i \leftarrow sr^{-1}$ // $y_i$ is the secret key!
          ∗ If there is no such record, sample $r \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, set $y_i \leftarrow sr^{-1}$, set $R \leftarrow g_{\mathsf{Sim}}^r$ and store $(\mathcal{H}_{\mathbb{G}}, pw, r, R)$
      • Record $(H, B^{y_i}, ssid, k)$ // Programming $H$ to the output key
      • Send $(\langle i, (pw, id, R, y_i, B)\rangle)_i \in \mathsf{state}$ to $\mathcal{A}$
```

Fig. 11. Simulation of adaptive corruptions for Π_{bCPace}.

7.1 Adaptive Security of EKE-NIKE

We considered the possibility of proving security of EKE-NIKE under adaptive corruptions assuming that the underlying sNIKE offers the standard notion of key corruption where the game reveals the long term secret key of an honest party, provided that no Test query has been made related to this key. However, the proof fails because in the UC setting adaptive corruption means that the internal state of a party may be revealed even if it has computed a key in the past.

For the particular case of standard PAKE discussed above, the proof can be extended for the adaptive corruption case, assuming that parties erase the secret keys after use. (This is good practice in general to achieve forward security in practice.) To see this, note that the reduction to one-time sNIKE would either be 1) simulating an honest party before corruption, in which case it can use Test and CorrReveal and it will never need to reveal the underlying secret key; 2) or simulating the honest party after corruption, in which case it already obtained the secret key from the sNIKE game.

7.2 Adaptive Security of CPace

We demonstrate that bare CPace is adaptively secure, i.e., Theorem 3 holds with respect to adaptive party corruptions. In the real world, upon adaptively corrupting a party, the shell hands the internal state consisting of all of the shell's records $\langle i, (pw, id, g, a, A)\rangle$ to the adversary, where $\mathcal{H}_{\mathbb{G}}(pw) = g$ and $A = g^a$ is the message produced upon input $(\mathsf{NewSession}, sid, i, pw, id, \perp)$ to the party while it was not yet corrupted.

The simulation of adaptive corruptions is given in Figure 11. In a nutshell, the strategy of the simulator is to (1) adjust the secret keys of adaptively corrupted party $\mathcal{P}$ to the passwords of all previous NewSession inputs, and to (2) adjust the random oracle $H()$ to the previous output keys of $\mathcal{P}$. For (1), the simulator leverages the $\mathcal{H}_{\mathbb{G}}$ trapdoors as follows: let $\mathcal{H}_{\mathbb{G}}(pw) = R$ denote the generator that $\mathcal{P}$ should have computed upon input $(\mathsf{NewSession}, ..., pw, ...)$, and let $S := g_{\mathsf{Sim}}^{s}$ denote the simulated message of $\mathcal{P}$ upon that input. Note that, upon adaptive corruption, it is possible that Sim already handed R to $\mathcal{A}$, since $\mathcal{A}$ is allowed to query $\mathcal{H}_{\mathbb{G}}$ on arbitrary values. Sim now needs to figure out which secret key y_i "explains" the message S, i.e., for which y_i it holds that $\mathcal{H}_{\mathbb{G}}(pw)^{y}_{i} = S$. We have $\mathcal{H}_{\mathbb{G}}(pw)^{y_i} = (g_{\mathsf{Sim}}^{r})^{y_i} = g_{\mathsf{Sim}}^{s}$ for $y_i = sr^{-1}$, and hence Sim can claim sr^{-1} to be the secret key of $\mathcal{P}$ computed upon receiving pw as input. For (2), we then use secret key y_i to compute the Diffie-Hellman value $k = B^{y_i}$ computed by $\mathcal{P}$ upon receiving message B.

In our security proof, we can insert the simulation strategy for (1) directly before game $\mathbf{G}_{11}$, which is the first game in which Sim skips hashing the passwords of the parties. We can insert the simulation strategy for (2) directly before game $\mathbf{G}_{8}$, which is the first game in which output keys are kept from Sim. Note that the programming is guaranteed to succeed because in game $\mathbf{G}_{7}$ it is excluded that $\mathcal{A}$ had previously submitted k to oracle H.

References

[ABB⁺20] Michel Abdalla, Manuel Barbosa, Tatiana Bradley, Stanislaw Jarecki, Jonathan Katz, and Jiayu Xu. Universally composable relaxed password authenticated key exchange. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 278–307. Springer, Heidelberg, August 2020.

[AHH20] Michel Abdalla, Bjorn Haase, and Julia Hesse. CPace, a balanced composable pake. IRTF CFRG draft, 2020.

[AHH21] Michel Abdalla, Björn Haase, and Julia Hesse. Security analysis of CPace. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part IV*, volume 13093 of *LNCS*, pages 711–741. Springer, Heidelberg, December 2021.

[AP05] Michel Abdalla and David Pointcheval. Simple password-based encrypted key exchange protocols. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 191–208. Springer, Heidelberg, February 2005.

[BBCW21] Manuel Barbosa, Alexandra Boldyreva, Shan Chen, and Bogdan Warinschi. Provable security analysis of FIDO2. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part III*, volume 12827 of *LNCS*, pages 125–156, Virtual Event, August 2021. Springer, Heidelberg.

[BCP⁺23] Hugo Beguinet, Céline Chevalier, David Pointcheval, Thomas Ricosset, and Mélissa Rossi. Get a CAKE: generic transformations from key encaspulation mechanisms to password authenticated key exchanges. In Mehdi Tibouchi and Xiaofeng Wang, editors, *Applied Cryptography and Network Security - 21st International Conference, ACNS 2023, Kyoto, Japan, June 19-22, 2023, Proceedings, Part II*, volume 13906 of *Lecture Notes in Computer Science*, pages 516–538. Springer, 2023.

[BCZ22] Nina Bindel, Cas Cremers, and Mang Zhao. FIDO2, CTAP 2.1, and WebAuthn 2: Provable security and post-quantum instantiation. Cryptology ePrint Archive, Report 2022/1029, 2022. `https://eprint.iacr.org/2022/1029`.

[BFK09] Jens Bender, Marc Fischlin, and Dennis Kügler. Security analysis of the PACE key-agreement protocol. In Pierangela Samarati, Moti Yung, Fabio Martinelli, and Claudio Agostino Ardagna, editors, *ISC 2009*, volume 5735 of *LNCS*, pages 33–48. Springer, Heidelberg, September 2009.

[BHvS12] Joseph Bonneau, Cormac Herley, Paul C. van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy*, pages 553–567. IEEE Computer Society Press, May 2012.

[BJX19] Tatiana Bradley, Stanislaw Jarecki, and Jiayu Xu. Strong asymmetric PAKE based on trapdoor CKEM. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 798–825. Springer, Heidelberg, August 2019.

[BM92] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *1992 IEEE Symposium on Security and Privacy*, pages 72–84. IEEE Computer Society Press, May 1992.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155. Springer, Heidelberg, May 2000.

[Can01a] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[Can01b] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *IEEE Symposium on Foundations of Computer Science – FOCS 2001*, pages 136–145. IEEE, 2001.

[Can19] Ran Canetti. Sids in uc-secure pake and ke. IRTF CFRG mail archive, 2019.

[CFR20] CFRG. Cfrg pake selection. IRTF website, 2020.

[CHK$^+$05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 404–421. Springer, Heidelberg, May 2005.

[CKS08] David Cash, Eike Kiltz, and Victor Shoup. The twin Diffie-Hellman problem and applications. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 127–145. Springer, Heidelberg, April 2008.

[CR03] Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 265–281. Springer, Heidelberg, August 2003.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[DHP$^+$18] Pierre-Alain Dupont, Julia Hesse, David Pointcheval, Leonid Reyzin, and Sophia Yakoubov. Fuzzy password-authenticated key exchange. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 393–424. Springer, Heidelberg, April / May 2018.

[FHKP12] Eduarda S.V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. Non-interactive key exchange. Cryptology ePrint Archive, Report 2012/732, 2012. https://eprint.iacr.org/2012/732.

[FHKP13] Eduarda S. V. Freire, Dennis Hofheinz, Eike Kiltz, and Kenneth G. Paterson. Non-interactive key exchange. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013*, volume 7778 of *LNCS*, pages 254–271. Springer, Heidelberg, February / March 2013.

[GdKQ+23] Phillip Gajland, Bor de Kock, Miguel Quaresma, Giulio Malavolta, and Peter Schwabe. Swoosh: Practical lattice-based non-interactive key exchange. Cryptology ePrint Archive, Report 2023/271, 2023. https://eprint.iacr.org/2023/271.

[GJK21] Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. KHAPE: Asymmetric PAKE from key-hiding key exchange. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 701–730, Virtual Event, August 2021. Springer, Heidelberg.

[GMR06] Craig Gentry, Philip MacKenzie, and Zulfikar Ramzan. A method for making password-based key exchange resilient to server compromise. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 142–159. Springer, Heidelberg, August 2006.

[Hes20] Julia Hesse. Separating symmetric and asymmetric password-authenticated key exchange. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 579–599. Springer, Heidelberg, September 2020.

[HGP+18] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. Rethinking access control and authentication for the home internet of things (IoT). In *27th USENIX Security Symposium (USENIX Security 18)*, pages 255–272, Baltimore, MD, August 2018. USENIX Association.

[HJK+18] Jung Yeon Hwang, Stanislaw Jarecki, Taekyoung Kwon, Joohee Lee, Ji Sun Shin, and Jiayu Xu. Round-reduced modular construction of asymmetric password-authenticated key exchange. In Dario Catalano and Roberto De Prisco, editors, *SCN 18*, volume 11035 of *LNCS*, pages 485–504. Springer, Heidelberg, September 2018.

[HJKW23] Julia Hesse, Stanislaw Jarecki, Hugo Krawczyk, and Christopher Wood. Password-authenticated TLS via OPAQUE and post-handshake authentication. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 98–127. Springer, Heidelberg, April 2023.

[HL17] Björn Haase and Benoît Labrique. Making password authenticated key exchange suitable for resource-constrained industrial control devices. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 346–364. Springer, Heidelberg, September 2017.

[HS14] Feng Hao and Siamak F. Shahandashti. The SPEKE protocol revisited. Cryptology ePrint Archive, Report 2014/585, 2014. https://eprint.iacr.org/2014/585.

[Jab97] David P. Jablon. Extended password key exchange protocols immune to dictionary attacks. In *6th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE 1997)*, pages 248–255, Cambridge, MA, USA, June 18–20, 1997. IEEE Computer Society.

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 456–486. Springer, Heidelberg, April / May 2018.

[KM15] Franziskus Kiefer and Mark Manulis. Oblivious pake: Efficient handling of password trials. In *Information Security*, pages 191–208, 2015.

[Kra03] Hugo Krawczyk. SIGMA: The "SIGn-and-MAc" approach to authenticated Diffie-Hellman and its use in the IKE protocols. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 400–425. Springer, Heidelberg, August 2003.

[KT11] Ralf Küsters and Max Tuengerthal. Composition theorems without pre-established session identifiers. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 41–50. ACM Press, October 2011.

[Mac01] Philip MacKenzie. On the security of the SPEKE password-authenticated key exchange protocol. Cryptology ePrint Archive, Report 2001/057, 2001. https://eprint.iacr.org/2001/057.

[PW17] David Pointcheval and Guilin Wang. VTBPEKE: Verifier-based two-basis password exponential key exchange. In Ramesh Karri, Ozgur Sinanoglu, Ahmad-Reza Sadeghi, and Xun Yi, editors, *ASIACCS 17*, pages 301–312. ACM Press, April 2017.

[SGJ23] Bruno Freitas Dos Santos, Yanqi Gu, and Stanislaw Jarecki. Randomized half-ideal cipher on groups with applications to UC (a)PAKE. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 128–156. Springer, Heidelberg, April 2023.

[SGJK22] Bruno Freitas Dos Santos, Yanqi Gu, Stanislaw Jarecki, and Hugo Krawczyk. Asymmetric PAKE with low computation and communication. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 127–156. Springer, Heidelberg, May / June 2022.

[Sho20] Victor Shoup. Security analysis of spake2+. Cryptology ePrint Archive, Paper 2020/313, 2020.

[W3C17] W3C. Web authentication working group. https://www.w3.org/groups/wg/webauthn/, 2017.

[Wik23] Wikpedia. Internet of things. https://en.wikipedia.org/wiki/Internet_of_things/, 2023.

A Details on Structured Protocols

We provide more details on the execution model for bare PAKE protocols that we use in this work, namely the concept of *structured protocols* [Can01a]. We briefly recall here the relevant aspects of the real and ideal world execution models, highlighting the aspects in which they differ from the typical presentation of UC proofs. We refer the reader to Section 3.3 for an overview.

Real world execution model. An environment $\mathcal{Z}$ is first invoked and, through its actions, creates a set of parties that will jointly execute a protocol, as well as an adversary $\mathcal{A}$. Parties, which are globally addressed by an *extended identifier*, are created when they receive a message/input for the first time.[20] The extended identifier is of the form $(sid, \mathcal{P}, \pi)$, where sid is a session identifier, $\mathcal{P}$ is a party identifier, and π is the protocol code. The action of creating the first party with extended identifier $(sid, \mathcal{P}, \pi)$ fixes the session identifier sid that defines the target session of the security analysis: all security notions are defined wrt to the behavior of parties carrying this session identifier. As usual, the environment is able to directly control the inputs and observe the outputs of parties. The environment can also communicate arbitrarily with adversary $\mathcal{A}$ that controls the communications between parties.[21]

We adopt the UC execution model for *structured* protocols, where the code of a party has two parts: a *shell* and a *body*. Intuitively, the body executes the cryptographic protocol, while the shell serves as a wrapper to that cryptographic protocol, dealing with session and state management, message passing, party addressing, etc. Such separation on the framework level perfectly serves the purpose of this work, namely analyzing key exchange from passwords only. Looking ahead, we will keep information such as party and session identifiers and tasks such as password instance management, the sending of messages, and the handling of incoming messages within the shell. The body then runs the bare PAKE protocol, accepting as input only a password at the beginning, and a message (or potentially multiple messages for multiple-round protocols) subsequently, and finally producing an output key. This strict separation allows us to analyze the security of key exchange from only passwords, ensuring on the framework level that none of the PAKE's guarantees depend on, e.g., uniqueness of session identifiers or common knowledge of party names. We now recap the formalism of separation into body and shell, which gives rise to so-called structured protocols.

Structured Protocols. The UC execution model for *structured* protocols, determines that the shell, which runs a wrapper around the bare PAKE protocol, receives inputs/messages under an extended identifier $(sid, \mathcal{P}, \pi)$ where session identifier, party identifier and code are each split into two parts, one corresponding to the body and the other corresponding to the shell.[22] Concretely we have $sid = (sid_{\mathrm{Sh}}, sid_{\mathrm{B}})$, $\mathcal{P} = (\mathcal{P}_{\mathrm{Sh}}, \mathcal{P}_{\mathrm{B}})$ and $\pi = (\pi_{\mathrm{Sh}}, \pi_{\mathrm{B}})$. The body, which runs

[20] The extended identifiers of environment $\mathcal{Z}$ and adversary $\mathcal{A}$ are restricted to values that exclude ambiguity and give rise to meaningful security notions. We omit the details.

[21] Without loss of generality we can assume that $\mathcal{A}$ is the dummy adversary that passes messages to/from parties from/to the environment $\mathcal{Z}$.

[22] It is common practice when presenting the syntax of UC protocols and the description of ideal functionalities to keep the code π implicit, and to assume that $\mathcal{P}$ and sid are given as inputs to the protocol. We also adopt this choice in the body of the paper, so as to keep notation aligned with the literature. However, this is a presentation choice and, formally, it is not what happens in the UC execution model. Indeed, $\mathcal{P}$ and sid are defined when the party machine is created, so that its extended identifier $(sid, \mathcal{P}, \pi)$ can be used to globally address it.

code π_{B}, only has access to $(sid_{\mathrm{B}}, \mathcal{P}_{\mathrm{B}})$, which in this work we will treat as *don't cares*: we arbitrarily fix them to $\bot$. We depict this separation in Figure 12. We emphasize that the session identifier is, as always in the UC setting, formally defining the context of the UC security analysis and, for this reason, it must be globally unique. Conversely, the party identifier is emulating an arbitrary party addressing mechanism within that particular protocol instance (it just needs to be different for every party running this protocol instance). One can think of it as any addressing scheme that is used in the real world, e.g., and IP address and a listening TCP port. What is crucial for the notion of bare PAKE is that neither of these parameters is directly accessible to the body of the cryptographic protocol, which captures the guarantee that the only input that really affects the correctness and security of a bare PAKE is the password.

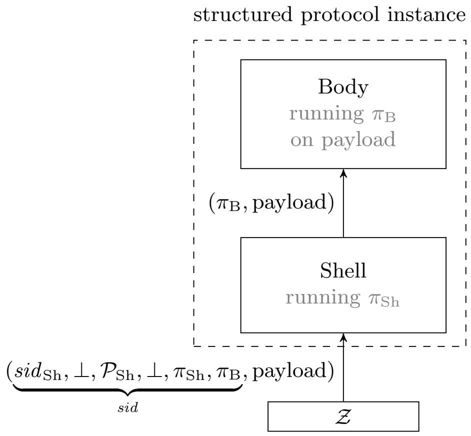

structured protocol instance
Body
running π_{B}
on payload
$(\pi_{\mathrm{B}}, \text{payload})$
Shell
running π_{Sh}
$(\underbrace{sid_{\mathrm{Sh}}, \bot, \mathcal{P}_{\mathrm{Sh}}, \bot, \pi_{\mathrm{Sh}}, \pi_{\mathrm{B}}}_{sid}, \text{payload})$
$\mathcal{Z}$

Fig. 12. Structured protocol with $sid_{\mathrm{B}} = \mathcal{P}_{\mathrm{B}} = \bot$, i.e., the body is receiving only the payload of the input and is oblivious of session and party information.

Ideal world execution model. The ideal world for structured protocols is the same as for unstructured protocols. When the environment creates a party, a dummy party is created instead. Dummy parties are actually using the ideal functionality that the real world protocol is compared to. Note that the dummy party is replacing the full real-world protocol $(\pi_{\mathrm{B}}, \pi_{\mathrm{Sh}})$. In particular, when the environment creates a new session at party $(\mathcal{P}, \bot)$ with session identifier $(sid, \bot)$, then the dummy party will inform the functionality of this fact. Conversely, outputs produced by the dummy party are triggered by a command issued by the ideal functionality. A simulator Sim (often called the ideal world adversary) must present to the environment a view that is indistinguishable from what $\mathcal{A}$ would provide in the real world. To achieve this, Sim has access to the backdoor interface of the ideal functionality.

```
On creation, generate and store $prm \leftarrow_{\mathrm{R}} \mathrm{Setup}$.

When $\mathcal{Z}$ passes input $(\mathsf{NewSession}, i, pw, id, \mathsf{role})$:
  ◦ ignore this query if record $\langle [i], \star \rangle$ exists
  ◦ if $\mathsf{role} \neq R$ (always the case for SSF protocols)
    – compute $(m, st_I) \leftarrow_{\mathrm{R}} \mathrm{StartIni}(prm, pw, id)$
    – store $\langle i, st_I \rangle$
    – send $(i, m)$ to $\mathcal{A}$
  ◦ else
    – compute $st_R \leftarrow_{\mathrm{R}} \mathrm{StartRsp}(prm, pw, id)$
    – store $\langle i, st_R \rangle$
    – send $i$ to $\mathcal{A}$ as coming from $(sid, \mathcal{P})$

When $\mathcal{A}$ delivers message $(i, m)$ to party running as $(sid, \mathcal{P})$:
  ◦ ignore this message if record $\langle [i], \star \rangle$ does not exist
  ◦ retrieve record $\langle [i], st \rangle$
  ◦ if $\mathsf{role} \neq R$ (always the case for SSF protocols)
    – compute $(K, ssid, cpid) \leftarrow \mathrm{EndIni}(st, m)$
    – on error, do nothing
    – otherwise, output $(K, ssid, cpname)$
  ◦ else
    – compute $(K, ssid, cpname, m') \leftarrow_{\mathrm{R}} \mathrm{EndRsp}(st, m)$
    – on error, do nothing
    – otherwise send $(i, m')$ to $\mathcal{A}$ and output $(K, ssid, cpname)$
```

Fig. 13. Shell code π_{Sh}. Works as a wrapper around body program of the form $\pi_{\mathrm{B}} = (\mathrm{Setup}, \mathrm{StartIni}, \mathrm{StartRsp}, \mathrm{EndIni}, \mathrm{EndRsp})$ and it runs under extended identifier $((sid_{\mathrm{Sh}}, \perp), (\mathcal{P}_{\mathrm{Sh}}, \perp), (\pi_{\mathrm{B}}, \pi_{\mathrm{Sh}}))$.

A Canonical Shell. A bare PAKE $(\mathrm{Setup}, \mathrm{StartIni}, \mathrm{StartRsp}, \mathrm{EndIni}, \mathrm{EndRsp})$ as specified above produces messages (and keys), but it does not contain any instructions on how to deal with communications, sending messages and processing incoming messages. For example, consider an application where a single party wants to produce keys from more than one password (e.g., a laptop that is set up to connect to various routers through WPA). Such a party runs several instances of StartIni, and needs to decide which of this instance an incoming message is processed with. We suggest a canonical shell code π_{Sh} in Fig. 13 that models what we find to be the most natural practical deployment of a bare PAKE protocol, and we will use it throughout this paper.

The shell π_{Sh} models an implementation that manages an arbitrary number of instances of the bare PAKE protocol by indexing them using an integer i. Here, an "instance of a bare PAKE protocol" corresponds to an input password, i.e., a single party runs as many instances as it obtains input passwords. An instance may be long-lived, cashing state and saving computation, or short-lived (e.g., used only once). Our framework allows both, and it is up to applications to choose a mode of operation that suits them. The instance identifier i can, for example, correspond to a TCP port that this instance uses to communicate. When a new instance i is created, the initialization code of the body is run, and the state is kept for future reference. The shell announces through the network adversary $\mathcal{A}$ that a new instance is running, along with any outgoing message that is already available for transmission. Incoming messages from the network are annotated with the target instance, which allows the shell to retrieve the correct state. Again, note that the same stored state can, according to this shell definition, be used to process multiple incoming messages, for example, a single initialization can be used to establish different keys with parties that share the same password. Note also that the shell π_{Sh} models applications where an instance must have been initialized (e.g., by creating a listening socket) before it can process any incoming messages.

Shell Variants: one-time passwords, message buffering, and more. π_{Sh} in Fig. 13 allows to address a specific password instance to compute a key with, without any restrictions. In particular, each password instance can be addressed multiple times and hence can produce multiple output keys. One could define arbitrary shell variants, in particular

- a "one-time" adaption of π_{Sh} where a state is invalidated after it is used to compute a key for the first time. We call this variant $\pi_{1\text{-}\mathrm{Sh}}$.
- a "buffering" adaption of π_{Sh} where incoming messages are recorded if the target instance does not yet exist, and are processed once the instance is initialized.
- a "flooding" adaption of π_{Sh} where incoming messages are processed with as many instances as possible, i.e., the maximum number of output keys is computed.

We will present detailed proofs for the canonical wrapper, and leave it to future work to extend the proofs to these other shell variants, which we expect to be straightforward.

B Details on Transformations

We start with the compiler from bare PAKE to PAKE, which is conceptually simpler.

B.1 Constructing PAKE from Bare PAKE

We show that any protocol that realizes the UC bare PAKE functionality (Figure 2) can be compiled in a straightforward manner to another one that realizes the multi-session variant of the original UC PAKE functionality $\mathcal{F}_{\mathsf{PAKE}}$ (Figure 1). For now, we only consider the non-relaxed version of both functionalities.

The compiler, depicted in Figure 14, takes the PAKE-conformant inputs sid, $\mathcal{P}$, $ssid_{\mathsf{PAKE}}$, $\mathcal{CP}$, and pw_{PAKE}, writes all these values into password pw_{bPAKE}, and then runs bPAKE using pw_{bPAKE}, using a fresh instance identifier i. The agreed secret key is simply the first session key produced by bPAKE instance i. Note that, intuitively, this transformation is delegating to bPAKE to make sure that a matching key can only be derived by a party that has matching PAKE-compliant inputs. The remaining features of bPAKE are not used, namely the ability to add/get party/coparty names, the ability to obtain session identifiers generated on the fly, and the ability to reuse a bPAKE instance.

On input $(\mathsf{NewSession}, sid, \mathcal{P}, ssid_{\mathsf{PAKE}}, \mathcal{CP}, pw_{\mathsf{PAKE}}, \mathsf{role})$, party $\mathcal{P}$ does:
- Set $pw_{\mathsf{bPAKE}} \leftarrow \{\mathcal{P}, \mathcal{CP}\}_{\mathsf{ord}} || sid || ssid_{\mathsf{PAKE}} || pw_{\mathsf{PAKE}}$.
- Take $i = ssid_{\mathsf{PAKE}}$ and store record $\langle i \rangle$.
- Send $(\mathsf{NewSession}, sid, i, pw_{\mathsf{bPAKE}}, \perp, \mathsf{role})$ to $\mathcal{F}_{\mathsf{bPAKE}}$.

On output $(sid, i, K, *, *)$ from $\mathcal{F}_{\mathsf{bPAKE}}$, party $\mathcal{P}$ does:
- Ignore this output if there was a previous output $(sid, i, *, *, *)$ by $\mathcal{F}_{\mathsf{bPAKE}}$.
- Ignore this output if record $\langle i \rangle$ does not exist.
- Output $(sid, ssid_{\mathsf{PAKE}}, K)$, where $ssid_{\mathsf{PAKE}} = i$.

Fig. 14. Transformation $\Pi_{\mathsf{bareP-to-P}}$ from bare PAKE to PAKE. Bare PAKE instance identifiers just need to be locally unique, so we reuse $ssid_{\mathsf{PAKE}}$ for that purpose.

Theorem 4 (BarePAKE-to-PAKE transformation). *The protocol $\Pi_{\mathsf{bareP-to-P}}$ realizes functionality $\mathcal{F}_{\mathsf{PAKE}}$ in the $\mathcal{F}_{\mathsf{bPAKE}}$-hybrid model. That is, for any efficient adversary $\mathcal{A}$ against $\Pi_{\mathsf{bareP-to-P}}$ (interacting with $\mathcal{F}_{\mathsf{bPAKE}}$), there exists an efficient simulator* Sim *that interacts with $\mathcal{F}_{\mathsf{PAKE}}$ and produces a view such that for all efficient environments it holds that*

$$\mathcal{D}_{\mathcal{Z}}^{\Pi_{\mathsf{bareP-to-P}}, \{\mathcal{F}_{\mathsf{PAKE}}, \mathsf{Sim}\}}(\kappa) \text{ is negligible in } \kappa,$$

where $\mathcal{D}_{\mathcal{Z}}^{\Pi_{\mathsf{bareP-to-P}}, \{\mathcal{F}_{\mathsf{PAKE}}, \mathsf{Sim}\}}(\kappa)$ is the distinguishing advantage of environment $\mathcal{Z}$ between the real world execution of $\Pi_{\mathsf{bareP-to-P}}$ and the simulation presented by Sim *interacting with $\mathcal{F}_{\mathsf{PAKE}}$.*

Proof Intuition. We provide a brief proof intuition. The core idea of the $\Pi_{\mathsf{bareP-to-P}}$ transformation is to embed PAKE-related information, such as the party identifiers $\mathcal{P}, \mathcal{CP}$, the unique subsession identifier $ssid_{\mathsf{PAKE}}$ for PAKE sessions into the "extended" password used by bPAKE parties. This fact can be used by the simulator to implement the ActiveNewKey and PassiveNewKey queries, which are used by the adversary to attack the underlying bPAKE protocol. Note that a password guess on the transformed protocol is only correct if all information embedded into the password matches, i.e., party identifiers and the unique session identifiers for the PAKE sessions need to match. The simulator hence always checks the "extended" part of the password but lets $\mathcal{F}_{\mathsf{PAKE}}$ decide whether the "regular" password is correct. The remainder of the proof consists mainly of bookkeeping to implement the other queries the adversary may issue. Figure 15 shows the full simulator.

Proof. We prove this theorem by constructing a sequence of games that step-wise transforms a real-world execution of the protocol $\mathit{\Pi}_{\mathsf{bareP-to-P}}$ (using $\mathcal{F}_{\mathsf{bPAKE}}$ as hybrid building block) into a simulated version of the protocol using the $\mathcal{F}_{\mathsf{PAKE}}$ functionality such that no efficient environment $\mathcal{Z}$ can distinguish the two worlds.

Game $\mathbf{G}_0$: The real execution. The environment $\mathcal{Z}$ runs the real protocol $\mathit{\Pi}_{\mathsf{bareP-to-P}}$ (using $\mathcal{F}_{\mathsf{bPAKE}}$ as hybrid building block) with adversary $\mathcal{A}$.

Game $\mathbf{G}_1$: Change layout. We move the whole execution of $\mathbf{G}_0$ into an ITI called simulator Sim. We add an empty ITI $\mathcal{F}$ that relays inputs and outputs of parties between $\mathcal{Z}$ and Sim, and we add dummy party relays between $\mathcal{Z}$ and $\mathcal{F}$ for each real party. The changes are only syntactical and hence we have

$$\Pr[\mathbf{G}_0] = \Pr[\mathbf{G}_1].$$

Game $\mathbf{G}_2$: Introduce the functionality. In this game we set $\mathcal{F} = \mathcal{F}_{\mathsf{PAKE}}$, with two modifications (which we will revert in later games):

(1) $\mathcal{F}$ relays all outputs of the form $(\mathit{ssid}_{\mathsf{PAKE}}, i, K)$ from Sim to $\mathcal{Z}$.

(2) $\mathcal{F}$ adds $\mathit{pw}_{\mathsf{PAKE}}$ to $\mathsf{NewSession}$ queries, i.e., it informs the simulator about passwords.

The changes are only syntactical because with modifications (1) and (2), all inputs and outputs are still relayed between the real execution within Sim and the environment $\mathcal{Z}$. The newly added interfaces $\mathsf{TestPwd}$ and NewKey are not yet called by the simulator and hence do not have any effect on the output distribution. We have

$$\Pr[\mathbf{G}_1] = \Pr[\mathbf{G}_2].$$

Game $\mathbf{G}_3$: Creating records. We change the simulator Sim as follows. Upon invocation it creates an empty set S', which it will later use to track partially and fully complete sessions. This set will later replace the internal set S the $\mathcal{F}_{\mathsf{bPAKE}}$ functionality uses. Additionally, the simulator implements the following:

- Whenever $\mathcal{F}_{\mathsf{PAKE}}$ outputs $(\mathsf{NewSession}, \mathit{sid}, \mathcal{P}, \mathit{ssid}_{\mathsf{PAKE}}, \mathcal{CP}, \mathit{pw}_{\mathsf{PAKE}}, \mathsf{role})$ the simulator Sim ignores the query if a record $\langle \mathcal{P}, \mathit{ssid}_{\mathsf{PAKE}}, \mathcal{CP}, \mathsf{role}\rangle$ already exists. Otherwise, it records $\langle \mathcal{P}, \mathcal{CP}, \mathit{ssid}_{\mathsf{PAKE}}, \mathsf{role}\rangle$ and sends $(\mathsf{NewSession}, \mathit{sid}, \mathcal{P}, \mathit{ssid}_{\mathsf{PAKE}}, \mathcal{CP}, \mathsf{role})$ to $\mathcal{A}$ (who expects this as a message from $\mathcal{F}_{\mathsf{bPAKE}}$). Additionally, the simulator suppresses any $\mathsf{NewSession}$ message from $\mathcal{F}_{\mathsf{bPAKE}}$ to any outside entity. Note that the interaction of Sim with the $\mathcal{F}_{\mathsf{bPAKE}}$ remains unchanged, i.e., it still sends $\mathsf{NewSession}$ message to $\mathcal{F}_{\mathsf{bPAKE}}$ such that the $\mathsf{ActiveNewKey}$ and $\mathsf{PassiveNewKey}$ queries made by $\mathcal{A}$ still work as intended.
- Whenever $(\mathsf{ActiveNewKey}, \mathit{sid}, \mathcal{P}, i, *, *, *)$ is queried by $\mathcal{A}$, the simulator ignores this query if the record $\langle \mathcal{P}, i, *, *\rangle$ already exists. Note that this disallows reusing a bPAKE instance, which is required by the PAKE functionality.

- Whenever $(\mathsf{PassiveNewKey}, sid, \mathcal{P}, i, \mathcal{CP}, i', ssid_{\mathsf{bPAKE}})$ is queried by $\mathcal{A}$, we ignore this query if either record $\langle \mathcal{P}, i, *, * \rangle$ or record $\langle \mathcal{P}', i', *, * \rangle$ (or both) already exist. Note that this disallows reusing a bPAKE instance, which is required by the PAKE functionality.

Note that whenever Sim would ignore incoming queries, the functionality $\mathcal{F}_{\mathsf{bPAKE}}$ would also ignore those queries. Hence, our changes do not influence the output distribution and we have

$$\Pr[\mathbf{G}_2] = \Pr[\mathbf{G}_3].$$

Game $\mathbf{G}_4$: $\mathcal{F}_{\mathsf{PAKE}}$ decides keys of honest sessions. In this game Sim implements the completion of honest sessions when the $(\mathsf{PassiveNewKey}, sid, \mathcal{P}, i, \mathcal{P}', i', ssid_{\mathsf{bPAKE}})$ query is issued by the adversary. First, Sim always checks if a record of form $\langle \mathsf{sesinf}, \mathcal{P}, i, ssid_{\mathsf{bPAKE}}, cpid \rangle$ (where $cpid$ can be extracted from the records filed in $\mathbf{G}_3$) exists. The simulator will create such record as soon as party $\mathcal{P}$ has output a key in protocol instance i. As such, the $\mathsf{PassiveNewKey}$ query is ignored, if a record of this form exists. The simulator implements the completion of honest session by distinguishing the following two cases:

(1) *The protocol participant who is first to finish.* First, Sim checks if $ssid_{\mathsf{bPAKE}} \notin \mathsf{S}'$, which only holds of neither $\mathcal{P}$ nor $\mathcal{P}'$ have completed the protocol yet. If this is the case, Sim adds $ssid_{\mathsf{bPAKE}}$ to S', tracking that one participant is now to complete the protocol. To this end, the simulator creates a record $\langle \mathsf{sesinf}, \mathcal{P}, i, ssid_{\mathsf{bPAKE}}, cpid \rangle$ and sends $(\mathsf{NewKey}, sid, \mathcal{P}, i, \perp)$ (where i corresponds to the PAKE session identifier) to $\mathcal{F}_{\mathsf{PAKE}}$, which outputs the correct key. Note that the newly created record allows Sim to check whether a participant has already completed its protocol run and output a key.

(2) *The protocol participant who is second to finish.* The second case is a little more involved. First, the simulator checks if a record of form $\langle \mathsf{sesinf}, \mathcal{P}', i', ssid_{\mathsf{bPAKE}}, id \rangle$ exists. Note that this check ensures that (i) P' has completed the protocol instance i' and (ii) that $\mathcal{P}$ expects to finish the protocol with id as partner. Only if this holds true, Sim records $\langle \mathsf{sesinf}, \mathcal{P}, i, ssid_{\mathsf{bPAKE}}, cpid \rangle$ (tracking that the second-to-finish participant will finish the protocol) and sends $(\mathsf{NewKey}, sid, \mathcal{P}, i, \perp)$ to $\mathcal{F}_{\mathsf{PAKE}}$, which results in the correct key being output.

(3) *"Extended" part of password is wrong.* Furthermore, Sim needs to cover a more subtle subcase. Note that, e.g., id in record $\langle \mathsf{sesinf}, \mathcal{P}', i', ssid_{\mathsf{bPAKE}}, id \rangle$ might not correspond to the name party $\mathcal{P}$ identifies itself with. In both cases, the simulator sends $(\mathsf{TestPwd}, sid, \mathcal{P}, i, \perp)$ and $(\mathsf{NewKey}, sid, \mathcal{P}, i, \perp)$ in sequence to $\mathcal{F}_{\mathsf{PAKE}}$, which results in outputting a random key towards $\mathcal{P}$, as required by a PAKE protocol.

If none of the above cases occur, the message is ignored by Sim. Note that by still forwarding the $\mathsf{PassiveNewKey}$ query to the original functionality $\mathcal{F}_{\mathsf{bPAKE}}$, we guarantee that the $\mathsf{ActiveNewKey}$ queries still works as intended. However, in order to guarantee a correct output distribution towards the adversary, we suppress all messages generated upon a $\mathsf{PassiveNewKey}$ query by $\mathcal{F}_{\mathsf{bPAKE}}$ but instead let Sim deliver messages as described above. We hence have

$$\Pr[\mathbf{G}_3] = \Pr[\mathbf{G}_4].$$

Game $\mathbf{G}_5$: $\mathcal{F}_{\mathsf{PAKE}}$ decides keys of attacked parties. In this game hop Sim implements the completion of attacked sessions when the $(\mathsf{ActiveNewKey}, sid, \mathcal{P}, i, pw^*_{\mathsf{bPAKE}}, K^*, ssid_{\mathsf{bPAKE}}, cpid)$ query is issued by the adversary. First, Sim always checks if a record of form $\langle \mathsf{sesinf}, \mathcal{P}, i, ssid_{\mathsf{bPAKE}}, cpid\rangle$ exists. The simulator will create such record as soon as party $\mathcal{P}$ has output a key in protocol instance i. As such, the $\mathsf{ActiveNewKey}$ query is ignored, if a record of this form exists. The simulator implements the completion of attacked session by distinguishing the following two cases:

(1) *The "extended" part of the password is wrong.* Recall that the transformed protocol uses passwords, where party identifiers, the PAKE subsession identifier, and the actual PAKE user password are embedded. Note that $\mathcal{F}_{\mathsf{PAKE}}$ can only check the PAKE user password, which leaves Sim to ensure that the other parts are correct as well. To this end, Sim parses $(\widetilde{\mathcal{P}}, \widetilde{\mathcal{P}'}, ssid_{\mathsf{PAKE}}, pw^*_{\mathsf{PAKE}}) \leftarrow pw^*_{\mathsf{bPAKE}}$, and checks if $\{\widetilde{\mathcal{P}}, \widetilde{\mathcal{P}'}\}_{\mathsf{ord}} \neq \{\mathcal{P}, \mathcal{P}'\}_{\mathsf{ord}}$ or $ssid_{\mathsf{PAKE}} \neq i$ hold. Should this be the case, the password guess by the adversary cannot be correct. Consequently, Sim sends $(\mathsf{TestPwd}, sid, \mathcal{P}, i, \bot)$ to $\mathcal{F}_{\mathsf{PAKE}}$, which will later result in a random key for $\mathcal{P}$.

(2) *The "extended" part of the password is correct.* If the check conducted in (1) was correct, then the "extended" part of the password guess pw^*_{bPAKE} must be correct. This leaves to verify, whether the PAKE user password is correct. To this end, the simulator sends $(\mathsf{TestPwd}, sid, \mathcal{P}, i, pw^*_{\mathsf{PAKE}})$ to $\mathcal{F}_{\mathsf{PAKE}}$. Note that (i) the functionality $\mathcal{F}_{\mathsf{PAKE}}$ will check if the guess pw^*_{PAKE} is correct and will react accordingly.

Finally, the simulator adds $ssid_{\mathsf{bPAKE}}$ to S', records $\langle \mathsf{sesinf}, \mathcal{P}, i, ssid_{\mathsf{bPAKE}}, cpid\rangle$, and sends $(\mathsf{NewKey}, sid, \mathcal{P}, i, K^*)$ to $\mathcal{F}_{\mathsf{PAKE}}$, which outputs a key towards party $\mathcal{P}$. Now that Sim already simulates $\Pi_{\mathsf{bareP-to-P}}$ with the aid of $\mathcal{F}_{\mathsf{PAKE}}$, it does not forward any more queries output by $\mathcal{F}_{\mathsf{bPAKE}}$. We have

$$\Pr[\mathbf{G}_4] = \Pr[\mathbf{G}_5].$$

Game $\mathbf{G}_6$: Remove password and key forwarding from functionality. As of $\mathbf{G}_5$, passwords of honest parties are not accessed by Sim anymore. At the same time, outputs of honest parties are produced by $\mathcal{F}_{\mathsf{PAKE}}$. We can hence remove the forwarding of passwords and keys that we temporarily installed in game $\mathbf{G}_5$, restoring the original $\mathcal{F}_{\mathsf{PAKE}}$ functionality, without modifying any outputs seen by $\mathcal{Z}$. We hence have

$$\Pr[\mathbf{G}_5] = \Pr[\mathbf{G}_6],$$

where the execution in this game is run by Sim of Figure 15 together with the PAKE functionality $\mathcal{F}_{\mathsf{PAKE}}$. This concludes the proof.

□

```
$(\mathbf{G}_3)$ Upon start, $\mathsf{Sim}$ initializes a set $\mathsf{S}' \leftarrow \{\}$ to track (partially and fully) completed sessions.

Messages from $\mathcal{F}_{\mathsf{PAKE}}$:

On $(\mathsf{NewSession}, sid, \mathcal{P}, ssid_{\mathsf{PAKE}}, \mathcal{CP}, \mathsf{role})$:
  – $(\mathbf{G}_3)$ Ignore if record $\langle \mathcal{P}, ssid_{\mathsf{PAKE}}, \mathcal{CP}, \mathsf{role} \rangle$ does exist.
  – $(\mathbf{G}_3)$ Record $\langle \mathcal{P}, ssid_{\mathsf{PAKE}}, \mathcal{CP}, \mathsf{role} \rangle$.
  – $(\mathbf{G}_3)$ Send $(\mathsf{NewSession}, sid, \mathcal{P}, ssid_{\mathsf{PAKE}}, \mathcal{CP}, \mathsf{role})$ to $\mathcal{A}$ as output of $\mathcal{F}_{\mathsf{bPAKE}}$.

Messages from $\mathcal{A}$:

On message $(\mathsf{LateTestPwd}, sid, \mathcal{P}, i, ssid_{\mathsf{bPAKE}}, pw^*_{\mathsf{bPAKE}})$ from $\mathcal{A}$:
  – Parse $(\widetilde{\mathcal{P}}, \widetilde{\mathcal{P}'}, ssid_{\mathsf{PAKE}}, pw^*_{\mathsf{PAKE}}) \leftarrow pw^*_{\mathsf{bPAKE}}$.
  – If $\{\widetilde{\mathcal{P}}, \widetilde{\mathcal{P}'}\}_{\mathsf{ord}} \neq \{\mathcal{P}, \mathcal{P}'\}_{\mathsf{ord}}$ or $ssid_{\mathsf{PAKE}} \neq i$:
      • Send $(\mathsf{LateTestPwd}, sid, i, \mathcal{P}, \perp)$ to $\mathcal{F}_{\mathsf{PAKE}}$.
  – Else:
      • Send $(\mathsf{LateTestPwd}, sid, i, \mathcal{P}, pw^*_{\mathsf{PAKE}})$ to $\mathcal{F}_{\mathsf{PAKE}}$.

On message $(\mathsf{ActiveNewKey}, sid, \mathcal{P}, i, pw^*_{\mathsf{bPAKE}}, K^*, ssid_{\mathsf{bPAKE}}, cpid)$ from $\mathcal{A}$:
  – $(\mathbf{G}_3)$ Ignore if record $\langle \mathcal{P}, i, [\mathcal{P}'], [\mathsf{role}] \rangle$ does not exist.
  – $(\mathbf{G}_5)$ Ignore if record $\langle \mathsf{sesinf}, \mathcal{P}, i, ssid_{\mathsf{bPAKE}}, cpid \rangle$ exists. // Key has already been output.
  – $(\mathbf{G}_5)$ If $ssid_{\mathsf{bPAKE}} \notin \mathsf{S}'$:
      • If $pw^*_{\mathsf{bPAKE}} \neq \perp$:
          ∗ $(\mathbf{G}_5)$ Parse $(\widetilde{\mathcal{P}}, \widetilde{\mathcal{P}'}, ssid_{\mathsf{PAKE}}, pw^*_{\mathsf{PAKE}}) \leftarrow pw^*_{\mathsf{bPAKE}}$.
          ∗ $(\mathbf{G}_5)$ [Guess is wrong.] If $\{\widetilde{\mathcal{P}}, \widetilde{\mathcal{P}'}\}_{\mathsf{ord}} \neq \{\mathcal{P}, \mathcal{P}'\}_{\mathsf{ord}}$ or $ssid_{\mathsf{PAKE}} \neq i$: Send $(\mathsf{TestPwd}, sid, \mathcal{P}, i, \perp)$ to $\mathcal{F}_{\mathsf{PAKE}}$.
          ∗ $(\mathbf{G}_5)$ [Guess may be correct.] Else: Send $(\mathsf{TestPwd}, sid, \mathcal{P}, i, \widetilde{pw})$ to $\mathcal{F}_{\mathsf{PAKE}}$
      • If $pw^*_{\mathsf{bPAKE}} = \perp$: Send $(\mathsf{RegisterTest}, sid, ssid_{\mathsf{PAKE}}, \mathcal{P})$ to $\mathcal{F}_{\mathsf{PAKE}}$.
      • $(\mathbf{G}_5)$ Set $\mathsf{S}' \leftarrow \mathsf{S}' \cup \{ssid_{\mathsf{bPAKE}}\}$.
      • $(\mathbf{G}_5)$ Record $\langle \mathsf{sesinf}, \mathcal{P}, i, ssid_{\mathsf{bPAKE}}, cpid \rangle$. // Record bPAKE session state
      • $(\mathbf{G}_5)$ Send $(\mathsf{NewKey}, sid, \mathcal{P}, i, K^*)$ to $\mathcal{F}_{\mathsf{PAKE}}$.

On message $(\mathsf{PassiveNewKey}, sid, \mathcal{P}, i, \mathcal{P}', i', ssid_{\mathsf{bPAKE}})$ from $\mathcal{A}$:
  – $(\mathbf{G}_3)$ Ignore if record $\langle \mathcal{P}, i, [cpid], [\mathsf{role}] \rangle$ does not exist.
  – $(\mathbf{G}_3)$ Ignore if record $\langle \mathcal{P}', i', *, [\mathsf{role}'] \rangle$ does not exist.
  – $(\mathbf{G}_4)$ Ignore if record $\langle \mathsf{sesinf}, \mathcal{P}, i, ssid_{\mathsf{bPAKE}}, cpid \rangle$ exists. // Key has already been output.
  – [Complete protocol for first participant.] If $ssid_{\mathsf{bPAKE}} \notin \mathsf{S}'$:
      • $(\mathbf{G}_4)$ Set $\mathsf{S}' \leftarrow \mathsf{S}' \cup \{ssid_{\mathsf{bPAKE}}\}$.
      • $(\mathbf{G}_4)$ Record $\langle \mathsf{sesinf}, \mathcal{P}, i, ssid_{\mathsf{bPAKE}}, cpid \rangle$. // Record bPAKE session state
      • $(\mathbf{G}_4)$ Send $(\mathsf{NewKey}, sid, \mathcal{P}, i, \perp)$ to $\mathcal{F}_{\mathsf{PAKE}}$.
  – [Complete protocol for second participant.] If there exists a record $\langle \mathsf{sesinf}, \mathcal{P}', i', ssid_{\mathsf{bPAKE}}, \mathcal{P} \rangle$:
      • $(\mathbf{G}_4)$ Record $\langle \mathsf{sesinf}, \mathcal{P}, i, ssid_{\mathsf{bPAKE}}, cpid \rangle$.
      • $(\mathbf{G}_4)$ Send $(\mathsf{NewKey}, sid, \mathcal{P}, i, \perp)$ to $\mathcal{F}_{\mathsf{PAKE}}$. // $\mathcal{F}_{\mathsf{PAKE}}$ decides if password is correct.
  – [“Extended” password is wrong.] If there exists a record $\langle \mathsf{sesinf}, \mathcal{P}, i', ssid_{\mathsf{bPAKE}}, * \rangle$:
      • $(\mathbf{G}_4)$ Send $(\mathsf{TestPwd}, sid, \mathcal{P}, i, \perp)$ to $\mathcal{F}_{\mathsf{PAKE}}$.
      • $(\mathbf{G}_4)$ Send $(\mathsf{NewKey}, sid, \mathcal{P}, i, \perp)$ to $\mathcal{F}_{\mathsf{PAKE}}$.
  – Ignore message otherwise.
```

Fig. 15. The simulator (without dashed boxes) constructed in the proof of Theorem 4. Lines prepended with $(\mathbf{G}_i)$ are added in game i. Dashed boxes can be included if a simulator for $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{LE}}}$ is desired.

On Supporting Lazy Extraction. We remark that we can also proof that $\Pi_{\mathsf{bareP-to-P}}$ realizes functionality $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{LE}}}$ in the $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$-hybrid model. The proof follows the same pattern as before but requires an additional game hop where the dashed boxes in Figure 15 are introduced.

B.2 Constructing Bare PAKE from PAKE

We show that any protocol that realizes the multi-session variant of the UC PAKE functionality (Fig. 1) can be compiled to another one that realizes $\mathcal{F}_{\mathsf{bPAKE}}$ (Fig. 2).

The intuition is that each key corresponding to a (non-repeat) bare PAKE output will be computed by a fresh PAKE session. The bare PAKE protocol we construct outlines a solution to the practical problem of defining a unique global session identifier and party identifiers that distinguish the participants, as required by PAKE. The protocol is shown in Figure 16. Observe that the $\mathcal{F}_{\mathsf{PAKE}}$ functionality is called with a random *ssid* agreed by the parties using two nonces, and that these nonces play a dual role: parties are using their nonce as their party identifier when calling $\mathcal{F}_{\mathsf{PAKE}}$. This dual use is not needed for particular applications that have another way of setting up the party identifiers used as input to the PAKE but here we are looking for a solution that does not rely on any external means to create the inputs to the PAKE.[23]

```
On input ($\mathsf{NewSession}, sid, \mathcal{P}, i, pw_{\mathsf{bPAKE}}, id, \mathsf{role}$):
  – Sample $N \leftarrow_{\mathrm{R}} \{0,1\}^{\kappa}$.
  – Store record $\langle i, pw_{\mathsf{bPAKE}}, id, \mathsf{role}, N\rangle$.
  – Send message $(sid, i, id, N)$ to the adversarial network.

On message $(sid, i, cpid, N')$ from the adversarial network:
  – Retrieve $\langle i, [pw_{\mathsf{bPAKE}}], [id], [\mathsf{role}], [N]\rangle$ (abort if instance $i$ does not exist).
  – Set $ssid \leftarrow \{N, N'\}_{\mathsf{ord}}$.
  – Set $pw_{\mathsf{PAKE}} \leftarrow \{id, cpid\}_{\mathsf{ord}}||pw_{\mathsf{bPAKE}}$.
  – Save $\langle ssid, i, cpid\rangle$.
  – Send ($\mathsf{NewSession}, sid, N, ssid, N', pw_{\mathsf{PAKE}}, \mathsf{role}$) to $\mathcal{F}_{\mathsf{PAKE}}$.

On output $(sid, ssid, K)$ from $\mathcal{F}_{\mathsf{PAKE}}$, party $\mathcal{P}$ does:
  – Retrieve $\langle ssid, [i], [cpid]\rangle$ and ignore if record does not exist.
  – Output $(sid, i, K, ssid, cpid)$.
```

Fig. 16. Transformation $\Pi_{\mathsf{P-to-bareP}}$ from PAKE to bare PAKE.

Theorem 5 (PAKE-to-barePAKE transformation). *The protocol in Figure 16 realizes functionality* $\mathcal{F}_{\mathsf{PAKE}}$ *in the* $\mathcal{F}_{\mathsf{bPAKE}}$*-hybrid model. That is, for any*

[23] Recall that party identifiers only need to be unique within the set of parties using a given session identifier.

efficient adversary $\mathcal{A}$ against $\mathit{\Pi}_{\mathsf{P-to-bareP}}$ (interacting with $\mathcal{F}_{\mathsf{PAKE}}$), there exists an efficient simulator Sim *that interacts with $\mathcal{F}_{\mathsf{PAKE}}$ and produces a view such that for all efficient environments it holds that*

$$\mathcal{D}_{\mathcal{Z}}^{\mathit{\Pi}_{\mathsf{P-to-bareP}},\{\mathcal{F}_{\mathsf{bPAKE}},\mathsf{Sim}\}}(\kappa) \leq 4q^2/2^\kappa,$$

where $\mathcal{D}_{\mathcal{Z}}^{\mathit{\Pi}_{\mathsf{P-to-bareP}},\{\mathcal{F}_{\mathsf{bPAKE}},\mathsf{Sim}\}}(\kappa)$ is the distinguishing advantage of environment $\mathcal{Z}$ between the real world execution of $\mathit{\Pi}_{\mathsf{P-to-bareP}}$ and the simulation presented by Sim *interacting with $\mathcal{F}_{\mathsf{bPAKE}}$, and where q is an upper bound on the number of protocol executions.*

Proof Intuition. We provide a brief proof intuition. The core idea of the $\mathit{\Pi}_{\mathsf{P-to-bareP}}$ transformation is to pre-agree on a unique subsession identifier *ssid* before the underlying PAKE protocol is executed. The subsession identifier consists of a concatenation of two nonces that have been contributed by the communicating parties. Note that those nonces serve a dual prupose in the transformation: (i) They serve to establish a unique subsession identifier for each session between two parties, and (ii) they also serve as party identities in the underlying PAKE protocol. That is, from a bPAKE perspective a user may identify with some name *id*, but the underlying PAKE protocol identifies the user via a nonce N instead. The simulator uses this as leverage to create multiple PAKE sessions for the same pair of bPAKE users. Additionally, the "extended" password pw' with added participant names allows the simulator to later distinguish whether both parties agree on the same names (i.e., if they do, both parties should get the same key as output, and if not, the keys should be random).The remainder of the proof consists mainly of bookkeeping to implement the other queries the adversary may issue. Figure 17 shows the full simulator.

Proof. We prove this theorem by constructing a sequence of games that step-wise transforms a real-world execution of the protocol $\mathit{\Pi}_{\mathsf{bareP-to-P}}$ (using $\mathcal{F}_{\mathsf{bPAKE}}$ as hybrid building block) into a simulated version of the protocol using the $\mathcal{F}_{\mathsf{PAKE}}$ functionality such that no efficient environment $\mathcal{Z}$ can distinguish the two worlds.

Game $\mathbf{G}_0$: The real execution. The environment $\mathcal{Z}$ runs the real protocol $\mathit{\Pi}_{\mathsf{P-to-bareP}}$ (using $\mathcal{F}_{\mathsf{bPAKE}}$ as hybrid building block) with adversary $\mathcal{A}$.

Game $\mathbf{G}_1$: Change layout. We move the whole execution of $\mathbf{G}_0$ into an ITI called simulator Sim. We add an empty ITI $\mathcal{F}$ that relays inputs and outputs of parties between $\mathcal{Z}$ and Sim, and we add dummy party relays between $\mathcal{Z}$ and $\mathcal{F}$ for each real party. The changes are only syntactical and hence we have

$$\Pr[\mathbf{G}_0] = \Pr[\mathbf{G}_1].$$

Game $\mathbf{G}_2$: Introduce the functionality. In this game we set $\mathcal{F} = \mathcal{F}_{\mathsf{bPAKE}}$, with two modifications (which we will revert in later games):

(1) If $\mathcal{F}$ relays all outputs of the form $(sid, i, K, ssid, cpid)$ from Sim to $\mathcal{Z}$.

(2) $\mathcal{F}$ adds pw to NewSession queries, i.e., it informs Sim about passwords.

Furthermore, Sim will not route messages of form (sid, i, id, N) (sent from Sim towards some party) and $(sid, i, cpid, N')$ (sent from the adversary on behalf of a some party towards an honest party $\mathcal{P}$) via $\mathcal{F}$, but rather relays them directly between the dummy parties and itself.

The changes are only syntactical because with modifications (1) and (2), all inputs and outputs are still relayed between the real execution within Sim and the environment $\mathcal{Z}$. The newly added interfaces $\mathsf{ActiveNewKey}$ and $\mathsf{PassiveNewKey}$ are not yet called by Sim and hence do not have any effect on the output distribution. We have

$$\Pr[\mathbf{G}_1] = \Pr[\mathbf{G}_2].$$

Game $\mathbf{G}_3$: First message for honest sessions. We change Sim Sim as follows:

- Whenever $\mathcal{F}_{\mathsf{bPAKE}}$ outputs $(\mathsf{NewSession}, sid, \mathcal{P}, i, id, \mathsf{role})$ alongside some password pw (due to $\mathbf{G}_5$), Sim behaves as follows. First it checks whether a record $\langle i, id, *, * \rangle$ exists, which would indicate that instance i of $\mathcal{P}$ has already taken part in some session. If so, Sim would ignore the query. Otherwise, it samples $N \leftarrow_{\mathrm{R}} \{0,1\}^{\kappa}$ according to the protocol specification, records $\langle i, id, \mathsf{role}, N \rangle$, and sends (sid, i, id, N) to $\mathcal{A}$ as a message from $\mathcal{P}$.
- The simulator does not change its interaction with the functionality $\mathcal{F}_{\mathsf{PAKE}}$ but suppresses all messages of above form from the original protocol run (which have been replaced by the messages generated above).

Since (1) our simulator only ignores queries that would also would have been ignored by the original protocol with $\mathcal{F}_{\mathsf{PAKE}}$ as building block, and (2) we did not introduce any new messages to the system but only replaced existing messages, our changes do not influence the output distribution and we have

$$\Pr[\mathbf{G}_2] = \Pr[\mathbf{G}_3].$$

Game $\mathbf{G}_4$: Abort upon nonce collision. In this game Sim tracks all nonces $N, N' \in \{0,1\}^{\kappa}$ generated by honest parties during the execution of the protocol. If any of the nonces collide (i.e., if N or N' were sampled twice in different sessions), then Sim aborts the simulation. Let $q \in \mathbb{N}$ be an upper bound on the number of protocol executions, then we have

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_4]| \leq 4q^2/2^{\kappa}.$$

Note that nonces for honest parties are now unique.

Game $\mathbf{G}_5$: Acting on second message. In this game we describe how Sim acts upon receiving the second message $(sid, i, cpid, N')$ for some honest party $\mathcal{P}$ from the adversary. The simulator first retrieves the record $\langle \mathcal{P}, i, [id], [\mathsf{role}], [N] \rangle$ to ensure that instance i of party $\mathcal{P}$ is indeed honest and exists. Should the record not exist, Sim will ignore the incoming message.

Next, Sim sets $ssid \leftarrow \{N, N'\}_{\mathsf{ord}}$ and distinguishes two cases: (i) N' was generated maliciously, and (ii) N' was generated honestly.

(i) *N' was generated maliciously.* Note that Sim can check this due to the changes made in game $\mathbf{G}_3$, i.e., whenever it receives a nonce N', it tries to retrieve a record of form $\langle *, *, *, *, N' \rangle$. If it does not exist, the nonce was not generated honestly. Furthermore note, that it is not efficiently possible for the environment to spawn an honest party such that it matches with a maliciously generated nonce, as this would require the adversary to guess which nonce will be chosen by the honest party.

    If the nonce was generated maliciously, Sim stores a record of form $\langle \mathcal{P}, N, i, N', \perp, ssid, id, cpid \rangle$. Note that the $i' = \perp$ entry will help Sim to track whether a session has a dishonest participant.

(ii) *N' was generated honestly.* Conversely to the above, Sim knows that a nonce was generated honestly, if record $\langle *, [i'], *, *, N' \rangle$ exists. Hence it will store a new record of form $\langle \mathcal{P}, N, i, N', i', ssid, id, cpid \rangle$. Note that $i' \neq \perp$, which indicated that this session is honest.

Finally, Sim sends $(\mathsf{NewSession}, sid, N, ssid, N', \mathsf{role})$ to the adversary as an output of $\mathcal{F}_{\mathsf{PAKE}}$. Note that the nonces have been used as party identifiers here.

Since (1) our simulator only ignores queries that would also would have been ignored by the original protocol with $\mathcal{F}_{\mathsf{PAKE}}$ as building block, and (2) we did not introduce any new messages to the system but only replaced existing messages, our changes do not influence the output distribution and we have

$$\Pr[\mathbf{G}_4] = \Pr[\mathbf{G}_5].$$

Game $\mathbf{G}_6$: Functionality decides keys of parties. We change Sim as follows.

- Whenever it receives a message $(\mathsf{TestPwd}, sid, \mathcal{P}, ssid, pw^*_{\mathsf{PAKE}})$, it ignores the query if a record $\langle \mathsf{pwguess}, sid, \mathcal{P}, ssid, * \rangle$ exists. Otherwise, it creates a record $\langle \mathsf{pwguess}, sid, \mathcal{P}, ssid, pw^*_{\mathsf{PAKE}} \rangle$.
- Whenever it receives a message $(\mathsf{NewKey}, sid, \mathcal{P}, ssid, K^*)$ it does the following. First, it retrieves the record $r = \langle \mathcal{P}, [i], [\mathcal{P}'], [i'], ssid, [id], [cpid] \rangle$, which must exist for any given pair of session with at least one honest participant. The simulator distinguishes the following three cases:

  (i) *A password guess exists.* This is recognized by trying to a record of form $\langle \mathsf{pwguess}, sid, \mathcal{P}, ssid, [pw^*_{\mathsf{PAKE}}] \rangle$. If it exists, the simulator parses $\{id', cpid'\}_{\mathsf{ord}} || pw^*_{\mathsf{bPAKE}} \leftarrow pw^*_{\mathsf{PAKE}}$ and checks if the "extended" part of the password is indeed equal to id, $cpid$ from record r. If this is not the case, then the password guess was hence wrong and Sim sets the password guess to $pw^*_{\mathsf{bPAKE}} = \perp$. Otherwise, pw^*_{bPAKE} is left as is. The simulator concludes this case by sending $(\mathsf{ActiveNewKey}, \mathcal{P}, i, pw^*_{\mathsf{bPAKE}}, K^*, ssid, cpid)$ to $\mathcal{F}_{\mathsf{bPAKE}}$ (i.e., the functionality decides if the "regular" part of the password guess is correct). Note that this results in a random key if the password guess was wrong, or in K^* if the guess was correct.

  (ii) *Both parties are honest.* This case is recognized by the functionality by checking if $i' \neq \perp$ holds. Recall that this only holds if both parties are honest (see $\mathbf{G}_5$). Hence, Sim sends $(\mathsf{PassiveNewKey}, \mathit{sid}, \mathcal{P}, i, \mathcal{P}', i', \mathit{ssid})$ to $\mathcal{F}_{\mathsf{bPAKE}}$ and lets the functionality determine the output key.

(iii) *The counterparty is malicious.* If $i' = \bot$ holds, one party is malicious. Since a malicious party does not have an instance identifier i' known to the functionality, a random output key is expected here. The simulator achieves this by sending $(\mathsf{ActiveNewKey}, \mathcal{P}, i, \bot, \bot, \mathit{ssid}, \mathit{cpid})$ to $\mathcal{F}_{\mathsf{bPAKE}}$. Note that the password guess has been set to $\bot$, which will always result in a wrong password guess and a random key.

– The simulator suppresses all outgoing messages from the original $\mathcal{F}_{\mathsf{PAKE}}$ functionality.

Note that the behavior of Sim is now independent of $\mathcal{F}_{\mathsf{PAKE}}$, but the output distribution has not changed. We have

$$\Pr[\mathbf{G}_5] = \Pr[\mathbf{G}_6].$$

Game $\mathbf{G}_7$: Remove password and key forwarding from functionality. As of $\mathbf{G}_6$, passwords of honest parties are not accessed by Sim anymore. At the same time, outputs of honest parties are produced by $\mathcal{F}_{\mathsf{bPAKE}}$. We can hence remove the forwarding of passwords and keys that we temporarily installed in game $\mathbf{G}_5$, restoring the original $\mathcal{F}_{\mathsf{bPAKE}}$ functionality, without modifying any outputs seen by $\mathcal{Z}$. We hence have

$$\Pr[\mathbf{G}_6] = \Pr[\mathbf{G}_7],$$

where the execution in this game is run by Sim of Figure 17 together with the bare PAKE functionality $\mathcal{F}_{\mathsf{bPAKE}}$. This concludes the proof.

□

On Supporting Lazy Extraction. We remark that we can also prove that $\mathit{\Pi}_{\mathsf{P-to-bareP}}$ realizes functionality $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$ in the $\mathcal{F}_{\mathsf{PAKE^{LE}}}$-hybrid model. The proof follows the same pattern as before but requires an additional game hop where the dashed boxes in Figure 17 are introduced.

C Proofs of the theorems in Section 5

C.1 Proof of Theorem 1

Proof. We prove this theorem via a sequence of three games, where the first game is the (m,n)-$\mathsf{OT\text{-}CKS}$ game when $b = 0$ and the third game is the (m,n)-$\mathsf{OT\text{-}CKS}$ when $b = 1$. We define $\Pr[\mathbf{G}_i]$ as the probability of the adversary winning in game $\mathbf{G}_i$. We therefore have that

$$\mathbf{Adv}^{(m,n)\text{-}\mathsf{OT\text{-}CKS}}_{\mathcal{A},\mathsf{H[sNIKE]}}(\kappa) = |\Pr[\mathbf{G}_0] - \Pr[\mathbf{G}_2]|\,.$$

Game $\mathbf{G}_0$: This game is the real (m,n)-$\mathsf{OT\text{-}CKS}$ game when $b = 0$.

```
Messages from $\mathcal{F}_{\mathsf{bPAKE}}$:

On $(\mathsf{NewSession}, sid, \mathcal{P}, i, id, \mathsf{role})$:
  – $(\mathbf{G}_3)$ Ignore if record $\langle \mathcal{P}, i, id, *, * \rangle$ exists.
  – $(\mathbf{G}_3)$ Sample $N \leftarrow_{\mathrm{R}} \{0,1\}^{\kappa}$.
  – $(\mathbf{G}_3)$ Store $\langle \mathcal{P}, i, id, \mathsf{role}, N \rangle$.
  – $(\mathbf{G}_3)$ Send $(sid, i, id, N)$ to $\mathcal{A}$.

Messages from $\mathcal{A}$:

On message $(sid, i, cpid, N')$ from $\mathcal{A}$ for honest party $\mathcal{P}$:
  – $(\mathbf{G}_5)$ Retrieve $\langle \mathcal{P}, i, [id], [\mathsf{role}], [N] \rangle$ and ignore, if record does not exist.
  – $(\mathbf{G}_5)$ Set $ssid \leftarrow \{N, N'\}_{\mathsf{ord}}$.
  – $(\mathbf{G}_5)$ If $\langle *, *, *, *, N' \rangle$ does not exist: // $N'$ was generated maliciously.
      • $(\mathbf{G}_5)$ Store $\langle \mathcal{P}, N, i, N', \perp, ssid, id, cpid \rangle$.
  – $(\mathbf{G}_5)$ Else:
      • $(\mathbf{G}_5)$ Retrieve $\langle *, [i'], *, *, N' \rangle$. // $N'$ was generated honestly.
      • $(\mathbf{G}_5)$ Store $\langle \mathcal{P}, N, i, N', i', ssid, id, cpid \rangle$.
  – $(\mathbf{G}_5)$ Send $(\mathsf{NewSession}, sid, N, ssid, N', \mathsf{role})$ to $\mathcal{A}$ as output of $\mathcal{F}_{\mathsf{PAKE}}$. // $N, N'$ are used as party identifiers here.

On message $(\mathsf{TestPwd}, sid, \mathcal{P}, ssid, pw^*_{\mathsf{PAKE}})$ from $\mathcal{A}$:
  – $(\mathbf{G}_6)$ Ignore, if record $\langle \mathsf{pwguess}, sid, \mathcal{P}, ssid, * \rangle$ exists.
  – $(\mathbf{G}_6)$ Record $\langle \mathsf{pwguess}, sid, \mathcal{P}, ssid, pw^*_{\mathsf{PAKE}} \rangle$. // $\mathcal{A}$ does not expect feedback.

  On message $(\mathsf{RegisterTest}, sid, ssid, \mathcal{P})$ from $\mathcal{A}$:
    – Ignore, if record $\langle \mathsf{pwguess}, sid, \mathcal{P}, ssid, * \rangle$ exists. // $\implies$ session not fresh
    – Retrieve $\langle \mathcal{P}, [i], *, *, ssid, *, * \rangle$ and ignore, if record does not exist.
    – Record $\langle \mathsf{latetest}, sid, \mathcal{P}, i, ssid \rangle$.

  On message $(\mathsf{LateTestPwd}, sid, ssid, \mathcal{P}, pw^*_{\mathsf{PAKE}})$ from $\mathcal{A}$:
    – Retrieve $\langle \mathsf{latetest}, sid, \mathcal{P}, [i], ssid \rangle$ and ignore, if record does not exist.
    – Retrieve $\langle \mathcal{P}, *, *, *, ssid, [id], [cpid] \rangle$
    – Parse $\{id', cpid'\}_{\mathsf{ord}} || pw^*_{\mathsf{bPAKE}} \leftarrow pw^*_{\mathsf{PAKE}}$.
    – If $\{id', cpid'\}_{\mathsf{ord}} \neq \{id, cpid\}_{\mathsf{ord}}$: Set $pw^*_{\mathsf{bPAKE}} = \perp$.
    – Send $(\mathsf{LateTestPwd}, sid, \mathcal{P}, i, ssid, pw^*_{\mathsf{bPAKE}})$ to $\mathcal{F}_{\mathsf{bPAKE}}$.

On message $(\mathsf{NewKey}, sid, \mathcal{P}, ssid, K^*)$ from $\mathcal{A}$:
  – $(\mathbf{G}_6)$ Retrieve $\langle \mathcal{P}, [i], [\mathcal{P}'], [i'], ssid, [id], [cpid] \rangle$ and ignore, if record does not exist.
  – $(\mathbf{G}_6)$ If $\langle \mathsf{pwguess}, sid, \mathcal{P}, ssid, * \rangle$ exists:
      • $(\mathbf{G}_6)$ Retrieve $\langle \mathsf{pwguess}, sid, \mathcal{P}, ssid, [pw^*_{\mathsf{PAKE}}] \rangle$.
      • $(\mathbf{G}_6)$ Parse $\{id', cpid'\}_{\mathsf{ord}} || pw^*_{\mathsf{bPAKE}} \leftarrow pw^*_{\mathsf{PAKE}}$.
      • $(\mathbf{G}_6)$ If $\{id', cpid'\}_{\mathsf{ord}} \neq \{id, cpid\}_{\mathsf{ord}}$: Set $pw^*_{\mathsf{bPAKE}} = \perp$. // "Extended" part of password wrong.
      • $(\mathbf{G}_6)$ Send $(\mathsf{ActiveNewKey}, \mathcal{P}, i, pw^*_{\mathsf{bPAKE}}, K^*, ssid, cpid)$ to $\mathcal{F}_{\mathsf{bPAKE}}$.
  – $(\mathbf{G}_6)$ If $i' \neq \perp$ : // counterparty is honest
      • $(\mathbf{G}_6)$ Send $(\mathsf{PassiveNewKey}, sid, \mathcal{P}, i, \mathcal{P}', i', ssid)$ to $\mathcal{F}_{\mathsf{bPAKE}}$.
  – $(\mathbf{G}_6)$ Else: Send $(\mathsf{ActiveNewKey}, \mathcal{P}, i, \perp, \perp, ssid, cpid)$ to $\mathcal{F}_{\mathsf{bPAKE}}$. // counterparty is not honest
```

Fig. 17. The simulator (without dashed boxes) constructed in the proof of Theorem 5. Lines prepended with $(\mathbf{G}_i)$ are added in game i. Dashed boxes can be included if a simulator for $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$ is desired.

Game $\mathbf{G}_1$: This game works as $\mathbf{G}_0$ but with the following modification: the Test oracle uses an independent random oracle H' (outside of the adversary's view) as a key derivation function.

Clearly, $\mathbf{G}_0$ and $\mathbf{G}_1$ are identical until attacker ever queries H on a pre-key that is computed by the Test oracle. We call this event E and therefore have

$$|\Pr[\mathbf{G}_0] - \Pr[\mathbf{G}_1]| \leq \Pr[E]\,.$$

We defer bounding the probability that E occurs to the end, as this is the most intricate part of the proof.

Game $\mathbf{G}_2$: This game is the real (m,n)-OT-CKS game when $b = 1$. The adversary's view in the two games is identical except when two test queries on the same public keys return two different session keys in $\mathbf{G}_1$: this never happens according to the rules of $\mathbf{G}_2$. If this event never happens, then the keys output by H' and revealed by Test to the attacker in $\mathbf{G}_1$ are identically distributed to those generated and revealed by the same oracle in $\mathbf{G}_2$. The probability that this event occurs can be bounded using a union bound over all test queries and observing that, for each of them, the event occurs with exactly the probability of a correctness error. This means that

$$|\Pr[\mathbf{G}_1] - \Pr[\mathbf{G}_2]| \leq q_T \cdot \delta\,.$$

Bounding event E. We conclude the proof by bounding the probability that E occurs. To do this, we must introduce a series of modifications in $\mathbf{G}_1$.

We first introduce game $\mathbf{G}_1^1$, which initially samples $i, j \leftarrow_{\mathrm{R}} [m]$ and sets a bad event E_1 if E occurs, for the first time, for the random oracle input $\mathsf{sNIKE.ShKey}(sk_i, pk_j)$ computed by the test oracle. Since i, j are sampled independently from the adversary's view, it is clear that

$$\Pr[E] \leq m^2 \cdot \Pr[E_1]\,.$$

At this point E_1 refers to only two key pairs, and it seems that a direct reduction to the one-wayness game is straightforward: the reduction can generate all honest key pairs except i and j, which it programs with the challenge keys from the one-wayness games. This is indeed the strategy we will use. However, there is still the possibility that the attacker places $\mathsf{CorrReveal}$ queries that require knowledge of sk_i or sk_j to answer. We make the following observations:

- The one-time use restriction implies that an honest key is either used in the test oracle, or in the corrupt reveal oracle.
- If E_1 occurs, then at least one of the two honest public keys pk_i or pk_j will be used in the test oracle.
- This implies that, worst case, the reduction will need to deal with one corrupt reveal query on either pk_i or pk_j.
- Fixing an honest key pk_1 and a corrupt key pk_2 in the input to $\mathsf{CorrReveal}$ fully determines the random oracle input associated with this query (as the shared key derivation algorithm is deterministic).

- The adversary can make at most q_H different queries to the random oracle, and at most one of them can be for the secret key that is computed by $\mathsf{CorrReveal}$ in that critical query. Let us call this the ℓ^*-th q_H query, and let $\ell^* = q_H + 1$ if the adversary never queries q_H on the input value that permits checking the consistency of this critical corrupt reveal query.

We now introduce game $\mathbf{G}_1^2$. This game initially samples a value $\ell \leftarrow_{\mathrm{R}} [q_H{+}1]$ and sets a bad event E_2 if E_1 occurs and $\ell^* = \ell$. Again, since ℓ is sampled independently of the adversary's view, we have that:

$$\Pr[E_1] \leq (q_H + 1) \cdot \Pr[E_2] \,.$$

Finally we can construct a reduction to the one-wayness game. It proceeds as we described above, but it only needs to offer a good simulation of game $\mathbf{G}_1^2$ if ℓ is correct. To perfectly simulate the critical corrupt-reveal query, in this case, it can simply use the output key of the ℓ-th query to H if $\ell \neq q_H + 1$; otherwise it uses a fresh random key. At the end of the adversary's run, the reduction chooses an entry K in the random oracle input log by sampling an index in the range $[q_H]$ and queries the guess oracle on (pk_i, pk_j, K). If E_2 occurred, this will be the correct answer to the OW challenge with probability $1/q_H$. We therefore have

$$\Pr[E_2] \leq q_H \cdot \mathbf{Adv}^{\mathsf{OW}}_{\mathcal{B},\mathsf{sNIKE}}(\kappa) \,.$$

The theorem follows by plugging together this inequality with the inequalities that relate games $\mathbf{G}_0$, $\mathbf{G}_1$ and $\mathbf{G}_2$.

□

C.2 Proof of Theorem 2

Proof. We present the simulator that justifies our protocol in Figure 8. The proof is structured as a sequence of games, which begins with the real-world experiment and terminates in the ideal world experiment where our simulator interacts with the $\mathcal{F}_{\mathsf{bPAKE}}$ functionality.

Game $\mathbf{G}_0$: This is the real-world experiment. We show the expanded protocol and operation of the ideal cipher (and the step-wise transformations during the next games) in Figure 18.

Game $\mathbf{G}_1$: We modify the operation of the ideal cipher, so that ciphertexts produced in the forward direction are sampled so as to be globally fresh across all queries placed to the ideal cipher. This game and $\mathbf{G}_0$ are identical until bad occurs, where bad is the event that a ciphertext is generated in $\mathbf{G}_0$ by computing the ideal cipher in the forward direction that collides with some pre-existing ciphertext at the output of the ideal cipher. This occurs with probability at most $q_{\mathsf{IC}}^2/|\mathcal{C}|$. The game now keeps a table mapping ciphertexts to passwords, that is updated whenever a new ciphertext is generated by the ideal cipher (this table, which we note fully identifies a password when given a ciphertext generated by the ideal cipher, will be useful in future games). We also note that this modification guarantees a ciphertext transmitted by an honest party fully identifies the session instance that transmitted it. We have

$$|\Pr[\mathbf{G}_0] - \Pr[\mathbf{G}_1]| \leq \Pr[\mathsf{bad}] \leq q_{\mathsf{IC}}^2/|\mathcal{C}|.$$

Game $\mathbf{G}_2$: We modify once more the operation of the ideal cipher, so that public keys generated in ideal cipher reverse queries are not sampled uniformly at random conditioned on the permutation property, but rather created using $\mathsf{sNIKE.Keygen}$ and aborting if the permutation property is violated. The ideal cipher table now also keeps the generated secret key.

Let $\epsilon_{\mathsf{sNIKE.Keygen}}$ be the maximum advantage of a ppt adversary in distinguishing public keys produced by $\mathsf{sNIKE.Keygen}$ from uniform. Then we can bound the distance between Games 1 and 2 as

$$|\Pr[\mathbf{G}_1] - \Pr[\mathbf{G}_2]| \leq q_{\mathsf{IC}}^2/|\mathcal{PK}| + q_{\mathsf{IC}} \cdot \epsilon_{\mathsf{sNIKE.Keygen}}$$

To see this, note that we can first modify Game 1 to sample uniform keys and aborting if a collision occurs, at the cost of a $q_{\mathsf{IC}}^2/|\mathcal{PK}|$ statistical distance. Then one can directly reduce the distance between the resulting two games to the pseudorandomness of $\epsilon_{\mathsf{sNIKE.Keygen}}$ using a hybrid argument over all reverse IC queries.

Game $\mathbf{G}_3$: At this point we replace all outputs of $\mathsf{sNIKE.ShKey}$ that are computed using two key pairs generated by the game with random values in $\mathcal{SK}$. Note the caching of secret keys to deal with consistency on matching sessions and delivery of repeat messages. Concretely, for sessions that agree on session keys computed from two ciphertexts that were generated by the game, it is guaranteed that the game itself generated the key pairs associated with these ciphertexts in the IC table: all such session keys are replaced with random ones. When the incoming ciphertext is not generated by the game, one must check if the correct password identifies a backward IC query, in which case the game also generated the associated key pair in the IC table. List L_K is therefore partitioned between those entries that contain two ciphertexts generated by the game (that intuitively correspond to passive or almost passive attacks where the attacker just changes the coparty names) and those entries that contain only one such ciphertext, and that correspond to active attacks with wrong password guesses (no two such entries will ever collide by lexicographic ordering, so there is no possibility of matching, as should be the case in active attacks).

We construct a reduction $\mathcal{B}$ that perfectly interpolates between $\mathbf{G}_2$ and $\mathbf{G}_3$ and wins the CKS game against the underlying sNIKE protocol with the same advantage that an attacker can distinguish the two games. The reduction simply uses the Test oracle to obtain the session computed between two public keys generated by the game, and $\mathsf{CorrReveal}$ to obtain the session keys that result from key agreements where one of the public keys was chosen by the adversary. We therefore have

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_2]| \leq \mathbf{Adv}_{\mathcal{B},\mathsf{sNIKE}}^{(q_{\mathsf{IC}},q_{\mathsf{IC}})\text{-}\mathsf{CKS}}(\kappa)\,,$$

where we bound the number of honest and corrupt public keys involved in the reduction indirectly using a bound on the number of ideal cipher calls executed during the game.

Game $\mathbf{G}_4$: In this game we replace the outputs of KDF with random keys whenever the input key to the function was randomly sampled by the game. This means that the KDF is now only computed when an active attack succeeds in guessing the correct password. Note that, at this point, the indexing of the table that keeps track of key matching and key repeats now uses *ssid*. This is consistent with the modification we introduced, since the KDF output is determined by its input key and *ssid*. The two games can be bridged using a hybrid argument over the KDF input keys involved in this change, where each hybrid step is a direct reduction to the PRF property of the KDF: note that the same KDF input key may be used over different *ssid* due to the possibility that the adversary manipulates the names of parties during transmission. If ϵ_{KDF} is the maximum advantage against the PRF property of the KDF by a ppt adversary, then we have

$$|\Pr[\mathbf{G}_3] - \Pr[\mathbf{G}_4]| \leq q_K \cdot \epsilon_{\mathsf{KDF}},$$

where q_K bounds the number of keys derived in the game.

Game $\mathbf{G}_5$: In this game we simply rearrange the code of the experiment to more closely match the split between the code of $\mathcal{F}_{\mathsf{bPAKE}}$ and the simulator in the ideal world. The only modification that introduces a difference in the adversary's view is that protocol sessions no longer generate and encrypt public keys, but rather directly sample a ciphertext for transmission as if a new forward IC query occurred. The public key of such a session is generated only if/when it is needed. This can happen in two cases: 1) the attacker decrypts this ciphertext under the correct password causing the key pair to be sampled; or 2) the attacker delivers another ciphertext to this session that decrypts, under the correct password, to a public key for which it may know the secret key. The secret key is only needed if the second case occurs and the game detects this event exactly like the simulator: the list S now contains only the ciphertexts and passwords corresponding to forward queries placed by the adversary, so it allows for password extraction; the game can therefore obtain the key pair by querying IC^{-1}. After this change, the execution of honest parties no longer relies on the password, except when checking if an active attacker is using the correct password, exactly matching the ideal world.

It remains to bound the probability that an attacker can distinguish $\mathbf{G}_5$ from $\mathbf{G}_4$. The two games are identical until bad' occurs, where bad' is the probability that one of two events occur: 1) $\mathbf{G}_4$ generates a repeat public key that does not cause a fresh ciphertext to be generated; 2) The attacker places a forward query to the IC that would have guessed an honest party's public key in $\mathbf{G}_4$, but this key has not yet been generated in $\mathbf{G}_5$ (in our pseudocode entries of the form $(\perp, \perp, \perp, c')$ in table T are assumed to never match adversarial input to the IC). Due to the lower bound on public-key min entropy, we have

$$|\Pr[\mathbf{G}_4] - \Pr[\mathbf{G}_5]| \leq 2 \cdot q_{\mathsf{IC}}^2 / 2^{\kappa}.$$

This concluded the proof. □

D Proof of Theorem 3

Proof. **Game $\mathbf{G}_0$: The real execution.** $\mathcal{Z}$ runs the real protocol $\mathit{\Pi}_{\mathsf{bCPace}}$ with adversary $\mathcal{A}$.

Game $\mathbf{G}_1$: Change layout. We move the whole execution of $\mathbf{G}_0$ into an ITI called simulator Sim. We add an empty ITI $\mathcal{F}$ that relays inputs and outputs of parties between $\mathcal{Z}$ and Sim, and we add dummy party relays between $\mathcal{Z}$ and $\mathcal{F}$ for each real party. The changes are only syntactical and hence we have

$$\Pr[\mathbf{G}_0] = \Pr[\mathbf{G}_1].$$

Game $\mathbf{G}_2$: Embedding trapdoors. In this game we let Sim sample a generator $g_{\mathsf{Sim}} \leftarrow_{\mathrm{R}} \mathbb{G}$ upon invokation. Sim then creates a record $(\mathcal{H}_{\mathbb{G}}, pw, x, Y)$ upon $\mathcal{H}_{\mathbb{G}}(pw)$ from $\mathcal{A}$ or any internally simulated honest party, where $x \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ is sampled at random and $Y = g_{\mathsf{Sim}}^x$ is sent as a reply to $\mathcal{A}$. Sim checks for existing records and replies consistently with the same Y if a record for some input x exists already. Since this way Sim implements a random group oracle, the changes are not noticable by the environment and we have

$$\Pr[\mathbf{G}_1] = \Pr[\mathbf{G}_2].$$

Game $\mathbf{G}_3$: Use generator g_{Sim} for the simulation of all honest parties. We change the simulation of honest parties as follows: upon each input $(\mathsf{NewSession}, sid, i, pw, id, \perp)$ and the simulated $\mathcal{P}$ querying $\mathcal{H}_{\mathbb{G}}(pw)$ with resulting record $(\mathcal{H}_{\mathbb{G}}, pw, x, h)$ and $\mathcal{P}$ sampling secret key $a \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and computing message $(sid, i, (h^a, id))$, Sim now creates an additional record $(\mathcal{P}, i, x \cdot a, h^a, id, \mathsf{keys}_i = [])$.

Since Sim does not make use of the new record, this change does not have any effect on the output distribution and is purely syntactical. We hence have

$$\Pr[\mathbf{G}_2] = \Pr[\mathbf{G}_3].$$

Note that $h^a = g_{\mathsf{Sim}}^{xa}$, i.e., with the change of this game, Sim now keeps track of exponents of honest parties' exponents w.r.t the "simulation base" g_{Sim}.

Game $\mathbf{G}_4$: Abort upon $\mathcal{H}_{\mathbb{G}}$ collision. In this game, we let Sim abort if it chooses $x \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ such that record $(\mathcal{H}_{\mathbb{G}}, *, x, *)$ already exists.

This and the previous game are equal except in case of an abort, which by the Birthday Bound is negligible, i.e.,

$$|\Pr[\mathbf{G}_4] - \Pr[\mathbf{G}_3]| \leq \frac{(q_{\mathsf{H2G}} + q_{\mathsf{var}})(q_{\mathsf{H2G}} + q_{\mathsf{var}} - 1)}{2 \cdot 2^{\kappa}},$$

Messages from $\mathcal{Z}$:

```
(G_0–G_4 only)
On $(\mathsf{NewSession}, sid, \mathcal{P}, i, pw, id, \mathsf{role})$:
  – Ignore if record $\langle \mathcal{P}, i, * \rangle$ exists or if $\mathsf{role} \neq \perp$.
  – $(sk, pk) \leftarrow_{\mathrm{R}} \mathsf{sNIKE.Keygen}(prm)$
  – $c \leftarrow \mathsf{IC}_{pw}(pk)$
  – $m \leftarrow (c, id)$
  – $st \leftarrow (sk, pw, m)$
  – Store $\langle \mathcal{P}, i, st \rangle$.
  – Send $(i, m)$ to $\mathcal{A}$.
```

```
(G_5 only)
On $(\mathsf{NewSession}, sid, \mathcal{P}, i, pw, id, \mathsf{role})$:
  – Ignore if record $\langle \mathcal{P}, i, * \rangle$ exists or if $\mathsf{role} \neq \perp$.
  – $c \leftarrow_{\mathrm{R}} \{c' \in \mathcal{C} \mid (\perp, \perp, \perp, c') \notin T\}$
  – Append $(\perp, \perp, \perp, c)$ to $T$.
  – $m \leftarrow (c, id)$
  – $st \leftarrow (\perp, pw, m)$
  – Store $\langle \mathcal{P}, i, st \rangle$.
  – Send $(i, m)$ to $\mathcal{A}$.
```

Messages from $\mathcal{A}$:

```
(G_0–G_4 only)
On message $(sid, i, m')$ from $\mathcal{A}$ towards honest $\mathcal{P}$:
  – Ignore if record $\langle \mathcal{P}, i, [st] \rangle$ does not exist.
  – Parse $(sk, pw, m) \leftarrow st$.
  – (G_3) Parse $(c, id) \leftarrow m$.
  – Parse $(c', id') \leftarrow m'$.
  – $ssid \leftarrow \{m, m'\}_{\mathsf{ord}}$
  – (G_3) If record $\langle [\mathcal{P}'], [i'], (\cdot, \cdot, m') \rangle$ or record $\langle [\mathcal{P}'], [i'], (\cdot, \cdot, (c', \cdot)) \rangle$ exists:
      (G_3 only)
      • If $\{c, c'\}_{\mathsf{ord}} \in L_K$, then: $\bar{K} \leftarrow L_K[\{c, c'\}_{\mathsf{ord}}]$.
      • Else: $\bar{K} \leftarrow_{\mathrm{R}} \mathcal{K}$.
      • $L_K[\{c, c'\}_{\mathsf{ord}}] \leftarrow \bar{K}$
      • $K \leftarrow \mathsf{KDF}(\bar{K}, ssid)$
      • (G_4) If $ssid \in L_K$, then: $K \leftarrow L_K[ssid]$.
      • (G_4) Else $K \leftarrow_{\mathrm{R}} \mathcal{K}$
      • (G_4) $L_K[ssid] \leftarrow K$
      • (G_3) Return $(K, ssid, id')$.
  – $pk' \leftarrow \mathsf{IC}_{pw}^{-1}(c')$
  – (G_3) If $(pw, [pk'], [sk'], c') \in T$ and $sk' \neq \perp$:
      (G_3 only)
      • If $\{c, c'\}_{\mathsf{ord}} \in L_K$, then: $\bar{K} \leftarrow L_K[\{c, c'\}_{\mathsf{ord}}]$.
      • Else: else $\bar{K} \leftarrow_{\mathrm{R}} \mathcal{K}$.
      • $L_K[\{c, c'\}_{\mathsf{ord}}] \leftarrow \bar{K}$.
      • $K \leftarrow \mathsf{KDF}(\bar{K}, ssid)$.
      • (G_4) If $ssid \in L_K$, then: $K \leftarrow L_K[ssid]$.
      • (G_4) Else: $K \leftarrow \mathcal{K}$
      • (G_4) $L_K[ssid] \leftarrow K$
      • (G_3) Return $(K, ssid, id')$.
  – $\bar{K} \leftarrow \mathsf{sNIKE.ShKey}(sk, pk')$
  – $K \leftarrow \mathsf{KDF}(\bar{K}, ssid)$
  – Output $(K, ssid, id')$.
```

```
(G_5 only)
On message $(sid, i, m')$ from $\mathcal{A}$ towards honest $\mathcal{P}$:
  – Ignore if record $\langle \mathcal{P}, i, [st] \rangle$ does not exist.
  – Parse $(\perp, pw, m) \leftarrow st$.
  – Parse $(c, id) \leftarrow m$.
  – Parse $(c', id') \leftarrow m'$.
  – $ssid \leftarrow \{m, m'\}_{\mathsf{ord}}$
  – If record $\langle [\mathcal{P}'], [i'], (\cdot, \cdot, m') \rangle$ or record $\langle [\mathcal{P}'], [i'], (\cdot, \cdot, (c', \cdot)) \rangle$ exists:
      • If $ssid \in L_K$, then: $K \leftarrow L_K[ssid]$.
      • Else $K \leftarrow_{\mathrm{R}} \mathcal{K}$
      • $L_K[ssid] \leftarrow K$
      • Return $(K, ssid, id')$.
  – If $L[c'] = \perp$:
      • If $ssid \in L_K$, then: $K \leftarrow L_K[ssid]$.
      • Else $K \leftarrow_{\mathrm{R}} \mathcal{K}$
      • $L_K[ssid] \leftarrow K$
      • Return $(K, ssid, id')$.
  – $(pw', pk') \leftarrow L[c']$
  – If $pw \neq pw'$ :
      • If $ssid \in L_K$, then: $K \leftarrow L_K[ssid]$.
      • Else: $K \leftarrow \mathcal{K}$
      • $L_K[ssid] \leftarrow K$
      • Return $(K, ssid, id')$.
  – $pk' \leftarrow \mathsf{IC}_{pw}^{-1}(c')$
  – Retrieve $(pw', pk, [sk], c)$ from $T$.
  – $\bar{K} \leftarrow \mathsf{sNIKE.ShKey}(sk, pk')$
  – $K \leftarrow \mathsf{KDF}(\bar{K}, ssid)$
  – Output $(K, ssid, id')$.
```

Ideal cipher calls:

```
On query $\mathsf{IC}_{pw}(pk)$:
  (G_0 only)
  – If $(pw, pk, [c]) \in T$, return $c$.
  – $c \leftarrow_{\mathrm{R}} \{c' \in \mathcal{C} \mid (pw, \cdot, c') \notin T\}$.
  – Append $(pw, pk, c)$ to $T$.
– (G_1) If $(pw, pk, \cdot, [c]) \in T$, return $c$.
– (G_1) $c \leftarrow_{\mathrm{R}} \{c' \in \mathcal{C} \mid (\cdot, \cdot, \cdot, c') \notin T\}$.
– (G_1) Append $(pw, pk, \perp, c)$ to $T$.
– (G_1) $L[c] \leftarrow (pw, pk)$.
– Return $c$.
```

```
On query $\mathsf{IC}_{pw}^{-1}(c)$:
  (G_0 only)
  – If $(pw, [pk], c) \in T$, return $pk$.
  – $pk \leftarrow_{\mathrm{R}} \{pk' \in \mathcal{PK} \mid (pw, pk', \cdot) \notin T\}$
  – Append $(pw, pk, c)$ to $T$.
– (G_2) If $(pw, [pk], \cdot, c) \in T$, return $pk$.
– (G_2) $(sk, pk) \leftarrow_{\mathrm{R}} \mathsf{sNIKE.Keygen}(prm)$
– (G_2) Abort if $(pw, pk, \cdot, \cdot) \in T$.
– (G_2) Append $(pw, pk, sk, c)$ to $T$.
– Return $pk$.
```

Fig. 18. Modifications of the expanded protocol and the ideal cipher during the sequence of games in the proof of Theorem 2. Dashed boxes are only included according to their description at the top.

The simulator ($\mathbf{G}_2$) samples a generator $g_{\mathsf{Sim}} \leftarrow_{\mathrm{R}} \mathbb{G}$. It also generates parameters for Π_{bCPace} by running $\mathrm{Setup}_{\mathsf{bCPace}}$ and hands them to any party querying $\mathcal{F}_{crs}$.

Messages from $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$

On $(\mathsf{NewSession}, sid, \mathcal{P}, i, \bot, id)$:
- ($\mathbf{G}_{11}$) Sample $s \leftarrow_{\mathrm{R}} \mathbb{Z}_p$, set $S \leftarrow g^s$
- ($\mathbf{G}_6$) Abort if s was sampled upon a $\mathsf{NewSession}$ query before
- ($\mathbf{G}_3$) Record $(\mathcal{P}, i, s, S, id, \mathsf{keys}_i = [])$, where keys_i is an empty array storing compromised keys output by $\mathcal{P}$'s instance i
- Send $(sid, i, (S, id))$ to $\mathcal{A}$ as message from $\mathcal{P}$

Messages from $\mathcal{A}$

On message $(sid, i, (X, cpid))$ from $\mathcal{A}$ towards honest $\mathcal{P}$:
- ($\mathbf{G}_8$) Retrieve record $(\mathcal{P}, i, [s, S, id, \mathsf{keys}_i])$, ignore message if no such record exists
- Set $ssid \leftarrow \{(S||id), (X||cpid)\}_{\mathsf{ord}}$
- ($\mathbf{G}_9$) For any tuple of records $(\mathcal{H}_{\mathbb{G}}, pw, r, R)$, $(\mathcal{H}_{\mathbb{G}}, pw', r', R')$ maintained by Sim, if $\mathcal{A}$ already submitted $(sid, H(K', ssid))$, $(sid, H(K, ssid))$ with $K = B^{s/r}$, $K' = B^{s/r'}$, then Sim aborts.
- ($\mathbf{G}_{10}$) If $(i, (X, cpid))$ is adversarially generated, do:
  - [$\mathcal{A}$ computed key already] If $\exists$ records $(\mathcal{H}_{\mathbb{G}}, [pw], r, *)$, $(H, K, ssid, [k])$ such that $K = X^{s/r}$: // Secret key of $\mathcal{P}$ is s/r, i.e., $CDH(H(pw), X, S) = CDH(g^r, X, g^s) = CDH(g, X, g^{s/r}) = X^{s/r}$.
    - $*$ Set $\mathsf{keys}_i[(X, cpid)] \leftarrow k$
    - $*$ Send $(\mathsf{ActiveNewKey}, sid, \mathcal{P}, i, pw, k, ssid, cpid)$ to $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$
  - [Adversarial $X \Rightarrow$ random key] If no such records exist, send $(\mathsf{ActiveNewKey}, sid, \mathcal{P}, i, \bot, \bot, ssid, cpid)$ to $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$.
- [Honest delivery] Else do:
  - ($\mathbf{G}_8$) Retrieve record $([\mathcal{CP}, i'], *, X, cpid, *)$ // Some party $\mathcal{CP}$ previously sent $(i, (X, cpid))$
  - ($\mathbf{G}_8$) Set $ssid \leftarrow \{(S||id), (X||cpid)\}_{\mathsf{ord}}$
  - ($\mathbf{G}_8$) Send $(\mathsf{PassiveNewKey}, sid, \mathcal{P}, i, \mathcal{CP}, i', ssid)$ to $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$.

Random oracle queries from $\mathcal{A}$

On query $\mathcal{H}_{\mathbb{G}}(pw)$ by $\mathcal{A}$:
- ($\mathbf{G}_2$) Retrieve record $(\mathcal{H}_{\mathbb{G}}, pw, *, [h])$ and reply with h
- ($\mathbf{G}_2$) Otherwise, do:
  - ($\mathbf{G}_2$) Sample $x \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, set $h \leftarrow g_{\mathsf{Sim}}^x$
  - ($\mathbf{G}_4$) Abort if record $(\mathcal{H}_{\mathbb{G}}, *, x, *)$ exists
  - ($\mathbf{G}_2$) Store $(\mathcal{H}_{\mathbb{G}}, pw, x, h)$ and reply with h.

On query $H(K, ssid)$ by $\mathcal{A}$:
- ($\mathbf{G}_7$) Retrieve record $(H, K, ssid, [k])$ and reply with k
- ($\mathbf{G}_7$) Otherwise, extract from $ssid$ values S_0, S_1, id_0, id_1
- ($\mathbf{G}_7$) If $\exists$ records $(*, *, s_0, S_0, *, *)$, $(*, *, s_1, S_1, *, *)$, $(\mathcal{H}_{\mathbb{G}}, *, x, *)$ with $K = g^{s_0 s_1/x}$ then abort // Guessed honest DH key
- ($\mathbf{G}_9$) For any three-tuple of records $(\mathcal{P}, i, s, S, id, \mathsf{keys}_i)$, $(\mathcal{H}_{\mathbb{G}}, pw, r, R)$, $(\mathcal{H}_{\mathbb{G}}, pw', r', R')$ maintained by Sim, if $\mathcal{A}$ already delivered message $(sid, i, (m, id'))$ to $\mathcal{P}$ and also submitted $(sid, H(K', ssid))$ with $K' = B^{s/r'}$: if $K = B^{s/r}$ and $ssid = \{(S||id), (m||id')\}_{\mathsf{ord}}$, then Sim aborts.
- ($\mathbf{G}_{10}$) [$\mathcal{A}$ computes key already given out to an honest party] If for any $j \in \{0,1\}$ $\exists$ records $(\mathcal{P}, [i], s_j, S_j, id, \mathsf{keys}_i)$ with $\mathsf{keys}_i[S_{1-j}, id_{1-j}] \neq \bot$ and record $(\mathcal{H}_{\mathbb{G}}, [pw], r, *)$ s.t. $K = B^{s_j/r}$:
  - Send $(\mathsf{LateTestPwd}, sid, \mathcal{P}, i, ssid, \mathsf{keys}_i[S_{1-j}, id_{1-j}])$ to $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$
  - Denote $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$'s answer to this query with k.
- In any other case, set $k \leftarrow_{\mathrm{R}} \{0,1\}^{\kappa}$
- ($\mathbf{G}_7$) Store $(H, K, ssid, k)$ and reply to $\mathcal{A}$ with k

Fig. 19. Simulator for bare CPace.

where q_{var} is the overall number of passwords in the system (note that it must hold that $q_{\mathsf{var}} \leq q_{\mathsf{ns}}$), and q_{H2G} is the number of $\mathcal{H}_{\mathbb{G}}$ queries issued by $\mathcal{A}$.

Game $\mathbf{G}_5$: Introduce the functionality. In this game we set $\mathcal{F} = \mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$ in the lazy-extraction version (including gray boxes), with two modifications: (1) $\mathcal{F}$ relays all outputs of the form $(sid, i, K, ssid, id)$ from Sim to $\mathcal{Z}$, (2) $\mathcal{F}$ adds pw to $\mathsf{NewSession}$ queries, i.e., informs the simulator about passwords. (All these modifications will be rolled back in later games.)

The changes are only syntactical because with modifications (1) and (2), all inputs and outputs are still relayed between the real execution within Sim and the environment $\mathcal{Z}$. The newly added interfaces $\mathsf{LateTestPwd}$, $\mathsf{ActiveNewKey}$ and $\mathsf{PassiveNewKey}$ are not yet called by the simulator and hence do not have any effect on the output distribution. We hence have

$$\Pr[\mathbf{G}_4] = \Pr[\mathbf{G}_5].$$

Game $\mathbf{G}_6$: Abort upon transcript collision In this game we let Sim abort if any honest party outputs a message $(sid, *, (A, *))$ where A was already output by an honest party before.

By the Birthday Bound, we have

$$|\Pr[\mathbf{G}_6] - \Pr[\mathbf{G}_5]| \leq \frac{q_{\mathsf{ns}}(q_{\mathsf{ns}} - 1)}{2 \cdot 2^{\kappa}},$$

where q_{ns} is the number of $\mathsf{NewSession}$ queries issued by $\mathcal{A}$. From this game on, messages produced by honest parties are unique.

Game $\mathbf{G}_7$: Abort upon DH key guess. We change the simulation of the H oracle as follows: upon receiving an query $H(K, ssid)$ from $\mathcal{A}$, Sim extracts from $ssid$ values $A||id, B||cpid$. If Sim finds records $(*, *, a, A, *, *)$, $(*, *, b, B, *, *)$, $(\mathcal{H}_{\mathbb{G}}, *, x, *)$ with $K = g_{\mathsf{Sim}}^{ab/x}$, Sim aborts.

This and the previous game produce equal outputs unless Sim aborts. We show that this happens only with negligible probability if the gapCDH assumption (Definition 3) holds in $\mathbb{G}$.

We build an attacker $\mathcal{B}_{\mathsf{gapCDH}}$ against the computational DH problem from a distinguisher $\mathcal{D}$ of games $\mathbf{G}_6$, $\mathbf{G}_7$. $\mathcal{B}_{\mathsf{gapCDH}}$ holds a challenge (g, A, B). $\mathcal{B}_{\mathsf{gapCDH}}$ randomly picks one $(\mathsf{NewSession}, [i, pw, \perp, id])$ query of $\mathcal{Z}$ and sets $\mathcal{H}_{\mathbb{G}}(pw) := g$, and $(sid, i, (A, id))$ as the outgoing message of the party $\mathcal{P}$ who received the $\mathsf{NewSession}$ input. Then, $\mathcal{B}_{\mathsf{gapCDH}}$ randomly picks another $(\mathsf{NewSession}, [i'], pw, \perp, [cpid])$ query of $\mathcal{Z}$ (i.e., with the same pw) to some party $\mathcal{P}' \neq \mathcal{P}$ and sets $(sid, i', (B, cpid))$ as the message of $\mathcal{P}'$. $\mathcal{B}_{\mathsf{gapCDH}}$ samples $K \leftarrow_{\mathrm{R}} \{0,1\}^{\kappa}$. If B gets delivered to instance $\mathcal{P}, i$ or/and A gets delivered to instance $\mathcal{P}', i'$, $\mathcal{B}_{\mathsf{gapCDH}}$ sets K to be the output by $\mathcal{P}$ or/and $\mathcal{P}'$.

We need to detail how $\mathcal{B}_{\mathsf{gapCDH}}$ computes the outputs of $\mathcal{P}$ and $\mathcal{P}'$ for other messages reaching them for their instances i and i'. Say message $(sid, i, (Z, id'))$ reaches instance $\mathcal{P}, i$, which had $(sid, i, (A, id))$ as a message. Since $\mathcal{B}_{\mathsf{gapCDH}}$ does not have secret key for instance $\mathcal{P}, i$, $\mathcal{B}_{\mathsf{gapCDH}}$, it will use its $\mathsf{DDH}(g, A, *, *)$ oracle to program H correctly: $\mathcal{B}_{\mathsf{gapCDH}}$ sets $ssid \leftarrow \{(A||id), (Z||id')\}_{\mathsf{ord}}$ and sets $(H(C, ssid), ssid, cpid)$ as the output of $\mathcal{P}$, where $\mathsf{DDH}(g, A, Z, C) = 1$. In case no such C can yet be found among $\mathcal{Z}$'s H queries, $\mathcal{B}_{\mathsf{gapCDH}}$ leaves a placeholder and fills in C as soon as $\mathcal{Z}$ sends it as an input to the random oracle H.

Outputs of $\mathcal{P}'$ are computed similarly, only that $\mathcal{P}'$ uses $H(pw') = g' \neq g$ and thus, oracle $\mathsf{DDH}(g', B, *, *)$ is required to compute the output keys of $\mathcal{P}'$.

If $\mathcal{Z}$ queries $H(K, s)$ with $s = \{(A||id), (B||cpid)\}_{\mathsf{ord}}$ and provides a solution $\mathsf{DDH}(g, A, B, K) = 1$, then $\mathcal{B}_{\mathsf{gapCDH}}$ submits K as CDH solution to its own challenge (g, A, B).

If $\mathcal{Z}$ never queries $H(K, \{(A||id), (B||cpid)\}_{\mathsf{ord}})$ with $\mathsf{DDH}(g, A, B, K) = 1$, then $\mathcal{B}_{\mathsf{gapCDH}}$ perfectly implements $\mathbf{G}_7$ and $\mathbf{G}_6$. This can be seen since the only change is in the implementation of the random oracle H using the DDH oracles, which is a perfect emulation of the H oracle in both games. It thus follows that

$$|\Pr[\mathbf{G}_7] - \Pr[\mathbf{G}_6]| \leq q_{\mathsf{var}} q_{\mathsf{pw}} \mathbf{Adv}_{\mathbb{G}}^{\mathsf{gapCDH}},$$

where q_{var} is the overall number of passwords in the system (note that it must hold that $q_{\mathsf{var}} \leq q_{\mathsf{ns}}$), and q_{pw} is the maximum number of parties receiving the same password through a NewSession input.

Game $\mathbf{G}_8$: Functionality decides keys of honest sessions. We change the simulation as follows. Whenever the adversary delivers an honestly generated message $(sid, i, (X, cpid))$ to some honest $\mathcal{P}$, Sim looks for records $(\mathcal{P}, i, *, [S, id], *)$, $([\mathcal{CP}, i'], *, X, cpid, *)$. If both exist, Sim sets $ssid \leftarrow \{(S||id), (X||cpid)\}_{\mathsf{ord}}$ and uses $\mathcal{F}_{\mathsf{bPAKE^{LE}}}$ to fix an output key for $\mathcal{P}$ by issuing a request $(\mathsf{PassiveNewKey}, sid, \mathcal{P}, i, \mathcal{CP}, i', ssid)$. Additionally, the output that the simulated $\mathcal{P}$ generated upon receipt of the message is suppressed.

Since every NewSession triggers a corresponding record created by Sim, Sim sends PassiveNewKey to instance $\mathcal{P}, i$ with partner $\mathcal{CP}, i'$ if and only if instance $\mathcal{P}, i$ outputs a key in $\mathbf{G}_7$ upon receipt of a message from $\mathcal{CP}, i'$. Because roles are all set to $\perp$ in CPace, and transcripts of honest parties are unique as of game $\mathbf{G}_6$, the PassiveNewKey query results in $\mathcal{P}$ to output a key for instance i. It is thus left to argue that this output by $\mathcal{P}, i$ is indistinguishable in both games. First, note that $ssid$ computed by Sim in $\mathbf{G}_8$ is equal to $ssid$ computed by $\mathcal{P}, i$ in $\mathbf{G}_7$. In $\mathbf{G}_7$, $\mathcal{P}$ computed the key as $H(K, ssid)$. In $\mathbf{G}_8$, $\mathcal{F}$ either repeats a key that was already output by $\mathcal{P}, i$ for the same $ssid$, or chooses $k \leftarrow_{\mathrm{R}} \{0,1\}^{\kappa}$ (in case 1.), or it uses a previously generated random key from a testable record (case 2.) — but in any case keys are generated freshly for each honest session. Since H is modeled as a random oracle, $\mathcal{H}_{\mathbb{G}}$ is collision-free (cf. game $\mathbf{G}_4$), and $\mathcal{Z}$ does not query H for DH keys computed by parties in honest sessions as of game $\mathbf{G}_7$, the output distribution does not change.

Game $\mathbf{G}_9$: Abort upon more than one password guess.

We let the simulator abort if $\mathcal{A}$ issues more than one password guess per actively attacked instance. Let us first describe how password guessing works in the real protocol Π_{bCPace}. Let $\mathcal{P}$ denote an honest party with input $(\mathsf{NewSession}, sid, i, pw, id, \bot)$. In game $\mathbf{G}_3$, a denotes $\mathcal{P}$'s secret key, let $\mathcal{H}_{\mathbb{G}}(pw) := g^x_{\mathsf{Sim}}$ and let $(\mathcal{P}, i, s := x \cdot a, A := g^s_{\mathsf{Sim}}, id, \mathsf{keys}_i)$ denote the corresponding record installed since game $\mathbf{G}_3$, where A is the message sent to the adversary $\mathcal{A}$. $\mathcal{A}$ can now attempt to guess pw by sending a message $(sid, i, (B, cpid))$ to $\mathcal{P}$, compute a key, and compare that key with the output key that $\mathcal{P}$ computes from B. This works as follows: $\mathcal{A}$ takes a password pw', samples a secret key $b \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and computes $B \leftarrow \mathcal{H}_{\mathbb{G}}(pw')^b$. Then $\mathcal{A}$ computes $k \leftarrow H(A^b, \{(A||id), (B||cpid)\}_{\mathsf{ord}}$. The honest party $\mathcal{P}$ that receives B computes and outputs $k' \leftarrow H(B^a, \{(A||id), (B||cpid)\}_{\mathsf{ord}})$. Because $\mathcal{H}_{\mathbb{G}}$ is collision free as of game $\mathbf{G}_4$, we have $(\mathcal{H}_{\mathbb{G}}(pw')^b)^a = B^a = A^b = (\mathcal{H}_{\mathbb{G}}(pw)^b)^a$ if and only if $pw = pw'$. Hence, $\mathcal{Z}$ can verify the password guess made by $\mathcal{A}$ by comparing k and k'.

With this game, we limit the allowed number of guesses per active attack (i.e., per adversarial B delivered to an honest instance $\mathcal{P}, i$) to 1. With the above, this means we let the simulator abort if the following happens: for any three-tuple of records $(\mathcal{P}, i, s, S, id, \mathsf{keys}_i)$, $(\mathcal{H}_{\mathbb{G}}, pw, r, R)$, $(\mathcal{H}_{\mathbb{G}}, pw', r', R')$ maintained by Sim, if $\mathcal{A}$ delivers message $(sid, i, (m, cpid))$ to $\mathcal{P}$ and also submits two queries $H(K, ssid)$, $H(K', ssid)$ with $K = B^{s/r}$, $K' = B^{s/r'}$ and $ssid = \{(m||id), (S||cpid)\}_{\mathsf{ord}}$, Sim aborts.

Clearly, if the abort does not happen, this and the previous game have an equal output distribution. We now show that the abort happens only with negligible probability if the simultaneous gapCDH assumption (Definition 4) holds in $\mathbb{G}$. $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ gets as input (g, g_1, g_2, g_3), sets $g_{\mathsf{Sim}} \leftarrow g$, randomly chooses two adversarial $\mathcal{H}_{\mathbb{G}}$ queries $\mathcal{H}_{\mathbb{G}}(pw), \mathcal{H}_{\mathbb{G}}(pw')$ with $pw \neq pw'$, stores $(\mathcal{H}_{\mathbb{G}}, pw, \bot, g_1), (\mathcal{H}_{\mathbb{G}}, pw', \bot, g_2)$ and replies to the adversary querying these oracles with g_1 and g_2. $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ also randomly picks a $\mathsf{NewSession}$ query $(\mathsf{NewSession}, sid, i, \tilde{pw}, id, \bot)$ with $\tilde{pw} \in \{pw, pw'\}$, which is input of some honest party $\mathcal{P}'$, and sets g_3 to be the resulting simulated message.

Let $\mathcal{P}$ denote an honest party receiving an input $(\mathsf{NewSession}, ..., \bar{pw}, ...)$, where $\bar{pw} \in \{pw, pw'\}$. We explain how $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ emulates the execution for the case $\bar{pw} = pw$, i.e., $\mathcal{H}_{\mathbb{G}}(\bar{pw}) = g_1$. $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ proceeds the execution for this $\mathsf{NewSession}$ query as in the previous game, i.e., lets $\mathcal{P}$ sample a secret key $a \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ and compute $A \leftarrow g_1^a$. The only difference is that $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ cannot record the $\mathcal{H}_{\mathbb{G}}$ trapdoor, and instead writes a record $(\mathcal{P}, i, \bot, A, id, \mathsf{keys}_i)$, leaving the third field empty.

Whenever a message $(sid, i, (m, cpid))$ gets delivered to $\mathcal{P}'$, as of game $\mathbf{G}_8$ no output key needs to be computed from it if that message is honest, because the key is generated by the functionality. $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ still needs to compute an output key if the message is adversarial though. In $\mathbf{G}_8$, the key was computed from the secret key of $\mathcal{P}'$, which is not known to $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ since g_3 was set as the message of $\mathcal{P}'$.

We distinguish two cases, depending on the password $\tilde{pw}$ input to $\mathcal{P}'$: (1) $\tilde{pw} = pw$, or (2) $\tilde{pw} = pw'$. $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ handles these cases similarly, using oracle $\mathsf{DDH}(g_1, g_3, *, *)$ for case (1) and $\mathsf{DDH}(g_2, g_3, *, *)$ for case (2).

We detail only case (1). On incoming message m, $\mathcal{P}'$ sets K as the output of $H(\mathsf{CDH}(g_1, g_3, m), \mathit{ssid})$. $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ perfectly emulates this behavior by using its DDH oracle $\mathsf{DDH}(g_1, g_3, m, K)$ on every query $H(K, \mathit{ssid})$, and if one such oracle query results in 1, $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ sets the hash output and the output key of $\mathcal{P}'$ to be equal. Note that this can happen in an arbitrary order: if $\mathcal{P}'$ needs to output the key before such a K is found, the output key is chosen at random by $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ and later programmed into H. Otherwise, the hash value is used as the output of $\mathcal{P}'$. (Note that the strategy for consistently computing party outputs is the same as in game $\mathbf{G}_7$.) Upon $\mathcal{A}$ querying $H(K, \mathit{ssid})$, $H(K', \mathit{ssid})$ with $\mathsf{DDH}(g_1, g_3, m, K) = 1$, $\mathsf{DDH}(g_2, g_3, m, K') = 1$ and $\mathit{ssid} = \{(m||\mathit{id}), (S||\mathit{cpid})\}_{\mathsf{ord}}$, $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ submits (m, K, K') as solution to its own experiment.

Since the missing entries in the records are not used, $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ perfectly emulates the execution of this game. Moreover, using its oracles, $\mathbf{Adv}_{\mathsf{sim-gapCDH}}$ can reliably detect solutions to its own challenge. We hence have

$$|\Pr[\mathbf{G}_9] - \Pr[\mathbf{G}_8]| \leq q_{\mathsf{H2G}}^2 q_{\mathsf{pw}} \mathbf{Adv}_{\mathbb{G}}^{\mathsf{sim-gapCDH}}.$$

Note: The need to emulate the behavior of other parties using the "challenge" passwords is unique to the bare PAKE setting, i.e., it does not occur when CPace is used as a standard PAKE [AHH21]. This is because in that version of CPace, ephemeral generators and secret keys are computed for each key exchange.

Game $\mathbf{G}_{10}$: Functionality decides keys of attacked parties. We change the simulation for the case where an attacked party outputs a key. Whenever the adversary delivers an adversarially generated message $(\mathit{sid}, i, (X, \mathit{cpid}))$ to some honest party $\mathcal{P}$, Sim tries to retrieve a record $(\mathcal{P}, i, [s, S, \mathit{id}, \mathsf{keys}_i])$ and drops the message if none exists. Otherwise, Sim continues by setting $\mathit{ssid} \leftarrow \{(S||\mathit{id}), (X||\mathit{cpid})\}_{\mathsf{ord}}$. Sim looks for records $(\mathcal{H}_{\mathbb{G}}, \mathit{pw}, r, R)$ and (H, K, ssid, k) generated from adversarial hashing requests such that $K = X^{s/r}$ (note: the $\mathcal{H}_{\mathbb{G}}$ record contains the adversarial password guess against the honest party, and the adversary already computed the output key k of that party). If these exist, then Sim updates $\mathsf{keys}_i([X, \mathit{cpid}]) \leftarrow k$ in the record and sends $(\mathsf{ActiveNewKey}, \mathit{sid}, \mathcal{P}, i, \mathit{pw}, k, \mathit{ssid}, \mathit{cpid})$ to $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$. Otherwise ($X$ is produced otherwise, $\mathcal{A}$ does not yet know the output key), Sim sends $(\mathsf{ActiveNewKey}, \mathit{sid}, \mathcal{P}, i, \perp, \perp, \mathit{ssid}, \mathit{cpid})$ to $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$.

At the same time, we need to treat the case where the attacker computes a key that was already output by an attacked party: we need to adjust the simulation of the H oracle, and let Sim program into it the keys output by the functionality towards attacked parties. We change the simulation of the H oracle on input $H(K, \mathit{ssid})$ by $\mathcal{A}$ as follows. Let $(S_0||\mathit{id}_0)$, $(S_1||\mathit{id}_1)$ denote the values extracted from ssid. Before sampling a random k as a reply to the H query of $\mathcal{A}$, Sim checks if for some $j \in \{0, 1\}$ $\exists$ records $(\mathcal{P}, [i], s_j, S_j, \mathit{id}_j, \mathsf{keys}_i)$ with $\mathsf{keys}_i[S_{1-j}, \mathit{id}_{1-j}] \neq \perp$ and record $(\mathcal{H}_{\mathbb{G}}, [\mathit{pw}], r, *)$ s.t. $K = B^{s_j/r}$ and $S_{1-j}, \mathit{id}_{1-j}$ is from an adversarially generated message. Sim sends $(\mathsf{LateTestPwd}, sid, \mathcal{P}, i, ssid, pw)$ to $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$. Upon reply k, Sim stores k in the H list and replies to $\mathcal{A}$ with k.

We need to argue indistinguishability for the outputs of $\mathcal{P}$ and the replies to adversarial H queries for the various cases of active attacks against $\mathcal{P}$.

(1) Output of $\mathcal{P}$ when attacked with valid password guess: if $\mathcal{P}$ used pw in instance i and Sim finds the records above, i.e., $\mathcal{A}$ already computed the key, then $\mathcal{P}$ outputs k in the previous game as well as in this game, since the $\mathsf{ActiveNewKey}$ query either passes k through to $\mathcal{P}$ in that case, or $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$ had already output k to $\mathcal{P}$ for the very same $i, ssid$ and now repeats that k to $\mathcal{P}$ (replaying a message results in the same key). If $\mathcal{P}$ uses a different password pw' in instance i, $\mathcal{P}$ outputs $H(\bar{K}, ssid)$ with $\bar{K} = X^{s/r'}$ for $\mathcal{H}_{\mathbb{G}}(pw') = g_{\mathsf{Sim}}^{r'}$ in the previous game and a key chosen at random by $\mathcal{F}$ in this game. This change is only noticeable if $\mathcal{A}$ submits $\bar{K}$ to the H oracle, which is excluded since game $\mathbf{G}_9$.

(2) Output of $\mathcal{P}$ when attacked without valid password guess: Consider now the case where Sim does not find two records $(\mathcal{H}_{\mathbb{G}}, pw, r, R)$ and $(H, K, ssid, k)$ such that $K = X^{s/r}$. The output of $\mathcal{P}$ is $H(K, ssid)$ with $\bar{K} = X^{s/r}$ and $\mathcal{H}_{\mathbb{G}}(pw) = g_{\mathsf{Sim}}^{r}$ for some pw that $\mathcal{P}$ received as input in the previous game. In this game, the output is a uniformly random value in this game, sampled by $\mathcal{F}$. If $\mathcal{A}$ does not submit K to H, the output is equally distributed. Otherwise, the execution is independent of r and hence $\mathcal{A}$ submits K to H only with negligible probability.

(3) Reply to H query, key found: In the previous game, $\mathcal{A}$ would receive as a reply the key output by $\mathcal{P}$ in instance i using pw, produced upon retrieval of adversarial message $(sid, i, (S_{1-j}, id_{1-j}))$, because that party computed its output as $H(K, ssid)$. In this game, Sim obtains some key k via $\mathsf{LateTestPwd}$ from $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$, where k was output by $\mathcal{P}$ in instance i using pw for session $ssid$ if and only if $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$ has a record $\langle \mathsf{latetest}, \mathcal{P}, ssid, pw, k \rangle$ flagged $\mathsf{testable}$. This is the case because the simulator in this game sent $(\mathsf{ActiveNewKey}, sid, \mathcal{P}, i, \perp, \perp, ssid, cpid)$ for the corresponding session of $\mathcal{P}$.

(4) Reply to H query, no key found: Since the simulation in this game only changes when a corresponding key is found in the records, the reply is still computed as in the previous game, i.e., random sampling of a reply to a fresh H query.

Game $\mathbf{G}_{11}$: Simulate messages without passwords. We change the simulation as follows: Upon query $(\mathsf{NewSession}, sid, \mathcal{P}, i, id, \perp)$ from $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$, Sim skips hashing pw and drawing a secret key a, and instead directly samples $s \leftarrow_{\mathrm{R}} \mathbb{Z}_q$. Sim then proceeds as in the previous game, i.e., computes the message of $\mathcal{P}$ as $S \leftarrow g_{\mathsf{Sim}}^{s}$ and records $(\mathcal{P}, i, s, S, id, \mathsf{keys}_i[])$.

Because $\mathbb{G}$ is a cyclic group, S of this game and of the previous game (where parties computed $S \leftarrow g_{\mathsf{Sim}}^{xa}$ with $a \leftarrow_{\mathrm{R}} \mathbb{Z}_q$) are equally distributed. Further, the simulation of this game does not use the secret key a of parties anymore, because the output of simulated parties is computed by $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$ (in honest sessions as of game $\mathbf{G}_8$, and in attacked sessions as of game $\mathbf{G}_{10}$). The $\mathcal{H}_{\mathbb{G}}$ trapdoor r of honest parties is also never accessed by Sim, since only $\mathcal{H}_{\mathbb{G}}$

records installed upon adversarial requests are retrieved. Hence, this and the previous game are equally distributed and we have

$$\Pr[\mathbf{G}_{10}] = \Pr[\mathbf{G}_{11}].$$

Game $\mathbf{G}_{12}$: Remove password and key forwarding from functionality. As of $\mathbf{G}_{11}$, passwords of honest parties are not accessed by the simulator anymore. At the same time, outputs of honest parties are produced by $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$. We can hence remove the forwarding of passwords and keys that we temporarily installed in game $\mathbf{G}_5$, restoring the original $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$ functionality, without modifying any outputs seen by $\mathcal{Z}$. We hence have

$$\Pr[\mathbf{G}_{11}] = \Pr[\mathbf{G}_{12}],$$

where the execution in this game is run by the simulator of Figure 19 together with the lazy-extraction bare PAKE functionality $\mathcal{F}_{\mathsf{bPAKE}^{\mathsf{LE}}}$.

□

D.1 Relating to previous analyses

Applying our barePAKE-to-PAKE transformation in Figure 14 to our bare CPace protocol Π_{bCPace} of Figure 9 yields a variant of CPace that is secure w.r.t the standard UC PAKE functionality (according to Theorem 4), and which can roughly be described as follows:

$$\begin{aligned} \text{generator} &\leftarrow \mathcal{H}_{\mathbb{G}}(pw, \{\mathcal{P}, \mathcal{CP}\}_{\mathsf{ord}}, ssid) \\ \text{final key} &\leftarrow H(k, A, B) \end{aligned}$$

"Basic CPace" of Abdalla et al. [AHH21], which is proven secure w.r.t $\mathcal{F}_{\mathsf{PAKE}^{\mathsf{SS}}}$, i.e., single-session PAKE, can be cast in a multi-session setting by applying the UC with Join State transformation [CR03]. This transformation adds session identifiers to all random oracles, and yields the following multi-session CPace variant:

$$\begin{aligned} \text{generator} &\leftarrow \mathcal{H}_{\mathbb{G}}(pw, \{\mathcal{P}, \mathcal{CP}\}_{\mathsf{ord}}, ssid) \\ \text{final key} &\leftarrow H(k, A, B, ssid) \end{aligned}$$

The only difference is hence the additional *ssid* in the final key derivation hash. Since adding this public identifier does not impact the security, applying our Theorem 4 to Theorem 3 confirms the standard UC security of basic CPace of Abdalla et al. [AHH21].

Next, we compare with the currently specified version of CPace at the IETF [AHH20]. Our bare CPace Π_{bCPace} of Figure 9 roughly does the following:

$$\begin{aligned} \text{generator} &\leftarrow \mathcal{H}_{\mathbb{G}}(pw) \\ \text{final key} &\leftarrow H(k, \{(A||id), (B||cpid)\}_{\mathsf{ord}}) \end{aligned}$$

Since the security of Π_{bCPace} is proven w.r.t arbitrary environments in Theorem 3, it holds in particular for *id* and *cpid* being empty strings. The resulting bare CPace

$$\begin{aligned} \text{generator} &\leftarrow \mathcal{H}_{\mathbb{G}}(pw) \\ \text{final key} &\leftarrow H(k, \{A, B\}_{\mathsf{ord}}) \end{aligned}$$

is that of [AHH20] when leaving out the optional session- and party identifiers from the specification. Our Theorem 3 hence shows that [AHH20] enjoys strong composability guarantees even without a unique session identifier, and without unique party identifiers that are known to both participants. Of course, without any identifying information of parties, such a variant of CPace might be of only limited applicability. It seems therefore that bare CPace is superior to the protocol specified in [AHH20], in the following sense:

- Bare CPace has composable security without unique and pre-exchanged session identifiers
- Both bare CPace and [AHH20] let parties learn authenticated information about the counterparty's identity, however
- Bare CPace does not put any conditions on party names $id, cpid$, while [AHH20] requires party identifiers to be pre-exchanged and unique among all protocol participants.