Smooth NIZK Arguments with Applications to Asymmetric UC-PAKE

Charanjit S. Jutla[1] and Arnab Roy[2]

[1] IBM T. J. Watson Research Center, Yorktown Heights, NY, USA
[2] Fujitsu Laboratories of America, Sunnyvale, CA

Abstract. We introduce a novel notion of smooth (-verifier) non- interactive zero-knowledge proofs (NIZK) which parallels the familiar notion of smooth projective hash functions (SPHF). We also show that the recent single group element quasi-adaptive NIZK (QA-NIZK) of Jutla and Roy (CRYPTO 2014) for linear subspaces can be easily extended to be computationally smooth. One important distinction of the new notion from SPHFs is that in a smooth NIZK the public evaluation of the hash on a language member using the projection key does not require the witness of the language member, but instead just requires its NIZK proof. This has the remarkable consequence that in the Gennaro-Lindell paradigm of designing universally-composable password-authenticated key-exchange (UC-PAKE) protocols, if one replaces the traditionally employed SPHFs with the novel smooth QA-NIZK, one gets highly efficient UC-PAKE protocols that are secure even under dynamic corruption. The new notion can be seen as capturing the essence of the recent UC-PAKE protocol of Jutla and Roy (AsiaCrypt 2015) which is secure under dynamic corruption but uses intricate dual-system arguments.
This simpler and modular design methodology allows us to give the first single-round asymmetric UC-PAKE protocol, which is also secure under dynamic corruption in the erasure model. Previously, all asymmetric UC-PAKE protocols required at least two rounds. In fact, our protocol just requires each party to send a single message asynchronously. In addition, the protocol has short messages, with each party sending only four group elements. Moreover, the server password file needs to store only one group element per client. The protocol employs asymmetric bilinear pairing groups and is proven secure in the (limited programmability) random oracle model and under the standard bilinear pairing assumption SXDH.

Keywords: QA-NIZK, PAKE, bilinear pairings, SXDH, MDDH, SPHF, hash proof, password, online attack, server compromise, dual system.

1 Introduction

Ever since the remarkably efficient non-interactive zero knowledge (NIZK) [BFM88] proofs for algebraic statements were developed by Groth and Sahai [GS08], there have been significant efficiency improvements and innovations in the construction of cryptographic protocols. Jutla and Roy [JR13, JR14] and Libert, Peters, Joye and Yung [LPJY14] further improved the efficiency of algebraic NIZK proofs, culminating in *constant* size NIZK proofs for linear subspaces, independent of the number of equations and witnesses. This efficiency improvement came in the weaker Quasi-Adaptive setting [JR13], which nevertheless proved sufficient for many applications.

Quasi-adaptive NIZK (QA-NIZK) proofs were further extended to provide simulation soundness [LPJY14, KW15] and dual-system simulation soundness [JR15], thus lending applicability to many more applications, such as, structure preserving signatures, password authenticated key exchange in the UC model and keyed homomorphic CCA-secure encryption.

In this paper, we further extend QA-NIZK proofs to provide an additional property called *smooth soundness*. The idea is to have a verifier that consists of three algorithms: a randomized hash key generation algorithm, a public hashing algorithm and a private hashing algorithm. The setting allows computation of a private hash given the private hash key and the word, while the public hash can be computed using the public or projection hash key and just a QA-NIZK argument for the word - neither the word nor the witness is required. Completeness states that the private hash is equal to the public hash for a language member and correct QA-NIZK proof. Computational soundness states that it is hard to come up with a proof such that a non-language word passes the same check. The new *smoothness* property states that for any non-language word, the private hash algorithm outputs a value (computationally) indistinguishable from uniform, even when the projection key is given to the adversary.

Comparison with SPHFs. The new primitive is modeled after smooth projective hash functions (SPHF [CS02]). An SPHF also generates public and private hash keys and defines a private hash and a public hash. Further, similar properties hold where (1) for a language member, private hash equals public hash, (2) for a non-language member, private hash is uniformly random. The crucial difference is that, whereas the SPHF public hash computation requires a witness of the language member, the smooth QA-NIZK public hash requires only a NIZK proof of the member. This allows for hiding of the witness, even when using the public hash key. On the other hand, our constructions only allow computational smooth-soundness, while for SPHFs these properties hold information theoretically.

Trapdoor SPHFs as introduced by [BBC$^+$13] allow a simulation world to have a trapdoor to evaluate a hash over a word without a witness and without having full access to the private hash key. This is different from smooth NIZK proofs which allow public hashing in the real world without a witness, but instead with a NIZK proof.

Password Authenticated Key Exchange. The problem of setting up a secure channel between two parties that only share a human-memorizable password (or a low-entropy secret) was first studied by Bellovin and Merritt [BM92], and Jiang and Gong [JG04]. Since then, this problem has been extensively studied and is called the *password-authenticated key-exchange* (PAKE) problem, while a protocol solving this problem is referred to as a PAKE protocol. Note that neither of the two parties is assumed to have a public key (for instance, if a public key infrastructure is not available or is considered insecure), and one of the main challenges in designing such protocols is the intricacy in the natural security definition which requires that the protocol transcripts cannot be used to launch *offline dictionary attacks*. While an adversary can clearly try to guess the (low-entropy) password and impersonate one of the parties, its advantage from the fact that the password is of low entropy should be limited to such *online* impersonation attacks. An example of an insecure protocol is one where the honest message flow includes a (deterministic) hash of the password, as then an adversary can launch an offline dictionary attack on the hash obtained from a single transcript.

In a subsequent paper, Bellovin and Merritt [BM93] also considered a stronger model of server compromise such that if a server's password file is revealed to the adversary it cannot directly impersonate a client (cf. if the password was stored in the raw at the server). The adversary should be able to impersonate the client only if it succeeds in an offline dictionary attack on the revealed server password file. Clearly, this requires that the server does not store the password as it is (or in some reversibly-encrypted form), and protocols satisfying this stronger security requirement are referred to as *asymmetric PAKE* protocols.

Canetti et al [CHK$^+$05] also considered designing (symmetric) UC-PAKE protocols in the universally-composable (UC) framework [Can01]. One of their main contributions was the definition of a natural UC-PAKE ideal functionality ($\mathcal{F}_{\text{PAKE}}$). Gentry et al [GMR06] extended the functionality of symmetric UC-PAKE [CHK$^+$05] to the asymmetric setting ($\mathcal{F}_{\mathsf{apwKE}}$) and gave a general method of extending any symmetric UC-PAKE protocol to an asymmetric UC-PAKE protocol (from now on referred to as UC-APAKE). Their general method adds an additional round to the UC-PAKE protocol. Moreover, their general two-round method requires that the environment somehow gets to know that in the UC-PAKE protocol both parties remain fresh, and this led them to define the functionality $\mathcal{F}_{\mathsf{apwKE}}$ to have additional TestAbort functions.

Our Contributions. In this paper, we give the first single-round UC-APAKE protocol (realizing $\mathcal{F}_{\mathsf{apwKE}}$). In fact, both parties just send a single message asynchronously. Since this is a single round protocol, we can realize $\mathcal{F}_{\mathsf{apwKE}}$ without the additional (and cumbersome) TestAbort function mentioned above. The protocol is realized in the (limited programmability [FLR$^+$10]) random-oracle [BR93] hybrid-model under standard static assumptions for bilinear groups, namely SXDH [BBS04] and the general MDDH assumption. Our protocol is also secure against adaptive corruption (in the erasure model) and is very succinct, with each message consisting of only four group elements. Moreover, for each client the server need store only *one* group elements as a "password hash". Many non-UC asymmetric PAKE protocols are at least two rounds [HK98, BPR00, BMP00, Mac01, Boy09]. Benhamouda and Pointcheval [BP13] proposed the first single round asymmetric PAKE protocol, but in a game-based model built on the BPR model [BPR00].

The first single-round UC-secure *symmetric* PAKE protocol was given in [KV11] (using bilinear pairings), which was then further improved (in the number of group elements) in subsequent papers [JR12, BBC$^+$13]. Recently [JR15], a single round UC-PAKE protocol (in the standard model and using bilinear pairings) was also proven secure against adaptive corruption using ideas from the dual-system IBE construction of Waters [Wat09]. However, the [JR15] construction did not employ their Dual-System Simulation Sound QA-NIZK proofs (DSS-QA-NIZK) in a black box manner. Instead, it used ideas from the DSS-QA-NIZK construction and properties as the underlying intuition for the proof.

In this paper, we show that the UC-PAKE of [JR15] can be built in a black box manner using smooth QA-NIZK arguments. The proof only uses the definitional properties of the smooth QA-NIZK, without referring to its specific construction.

Next, we build on the Verifier-based PAKE (VPAKE) construction of [BP13], to construct a new single round UC-APAKE protocol. The intuition behind VPAKEs is as follows. Clearly, the server has to store some form of encryption or (probabilistic or deterministic) hash of the password, so that an adversary on obtaining this server password file has to, at the very least, perform offline tests to recover the password. It is not difficult to see that offline tests suffice as the following argument shows: consider an adversary that has obtained this hash of the password from the server password file. Next, it impersonates the client by guessing a password pw', and impersonates the server using the hash of the password that it has obtained, and checks if both ends compute the same session key to verify if pw' was the correct guess.

Unfortunately, in the UC framework, the simulator has to detect these offline password guesses by an adversary which steals the server password file, and for provable security this seems to inevitably require the random oracle model. Non-UC asymmetric PAKE protocols, do not suffer from the same drawback. In fact, the focus of [BP13] was to propose a security definition and constructions which could be proven secure in the standard model.

In our protocol, each party sends an ElGamal style encryption of the (hash of) the password pw to the other party, along with an SPHF of the underlying language and a projection verification hash key of a smooth QA-NIZK of the underlying language (augmented with the SPHF). If such a message is adversarially inserted, the simulator must have the capability to extract password pw' from it, so that it can feed the ideal functionality $\mathcal{F}_{\mathsf{apwKE}}$ to test this guess of the password. Thus, the NIZK proof must have simulation-sound extractability. It was shown in [JR15] that dual-system simulation soundness suffices for this purpose (and that makes the protocol very simple). When using smooth QA-NIZK, this dual-system simulation-soundness can be attained by simply sending an SPHF (see above).

More details can be found in Sections 6.2 and 6.3, where we also explain how the random oracle is used to extract the password efficiently from the exponent.

This leads to a security reduction which has an additive computational overhead of $n{*}m{*}\text{poly}(q)$, where n is the number of random oracle calls, m is the number of online attacks and q is the security parameter. We remark that the random oracle model uses only limited programmability as defined by Fischlin et al. [FLR$^+$10]. Basically, the output values of the random oracle are all randomly chosen, but different inputs can be assigned dynamically to these outputs.

The rest of the paper is organized as follows. In Section 2 we recall SPHFs. In Section 3 we introduce the new notion of smooth QA-NIZK proofs. In Section 4, we recall the MDDH assumptions and establish a useful technical theorem relating the assumptions. In Section 5, we give the single group element smooth QA-NIZK construction for linear subspaces. In Section 6, we describe the ideal functionality $\mathcal{F}_{\mathsf{apwKE}}$ for asymmetric password-authenticated key-exchange, construct the new single-round UC-APAKE protocol and provide its proof of UC-realization of ideal functionality $\mathcal{F}_{\mathsf{apwKE}}$.

2 Preliminaries: Smooth Projective Hash Functions

Since we are interested in distributions of languages, we extend the usual definition of smooth projective hash functions (SPHFs) [CS02] to distribution of languages. So consider a parametrized class of languages $\{L_\rho\}_{\rho \in \mathsf{Lpar}}$ with the parameters coming from an associated parameter language Lpar. An SPHF consists of the following efficient algorithms.

- $\mathsf{hkgen}(\rho)$, which generates two keys, a private key called hk, and a public key called hp.
- $\mathsf{privH}(\mathsf{hk}, x, l)$, computes a hash (in set Π) using the private key on input word x and label l.
- $\mathsf{pubH}(\mathsf{hp}, x, l; w)$ computes a hash (in set Π) using the public key on an input word x with witness w (for language L_ρ) and label l.

The correctness of SPHF family states that for all languages L_ρ in the parametrized class, for all $x \in L_\rho$ (with witness w), and for all labels l,

$$\mathsf{privH}(\mathsf{hk}, x, l) = \mathsf{pubH}(\mathsf{hp}, x, l; w).$$

A projective hash function family is called **smooth** if for all $x \notin L_\rho$, $\mathsf{privH}(\mathsf{hk}, x, l)$ is statistically indistinguishable from a random element in Π, even given hp. It is called $\mathbf{smooth_2}$ if for all $x \notin L_\rho$, $\mathsf{privH}(\mathsf{hk}, x, l)$ is statistically indistinguishable from a random element in Π, even given hp and one evaluation of $\mathsf{privH}(\mathsf{hk}, x^*, l^*)$ for any $(x^*, l^*) \neq (x, l)$.

3 Smooth Quasi-Adaptive NIZK Proofs

We start by reviewing the definition of Quasi-Adaptive computationally-sound NIZK proofs (QA-NIZK) [JR13]. A witness relation is a binary relation on pairs of inputs, the first called a (potential) language member and the second called a witness. Note that each witness relation R defines a corresponding language L which is the set of all x for which there exists a witness w, such that $R(x, w)$ holds.

We will consider QA-NIZK proofs for a probability distribution $\mathcal{D}$ on a collection of (witness-) relations $\mathcal{R} = \{R_\rho\}$ (with corresponding languages L_ρ). Recall that in a QA-NIZK, the CRS can be set after the language parameter has been chosen according to $\mathcal{D}$. Please refer to [JR13] for detailed definitions.

Definition 1. *([JR13]) We call* $(\mathsf{pargen}, \mathsf{crsgen}, \mathsf{prover}, \mathsf{ver})$ *a* (labeled) quasi-adaptive non-interactive zero-knowledge *(QA-NIZK) proof system for witness-relations* $\mathcal{R}_\lambda = \{R_\rho\}$ *with parameters sampled from a distribution* $\mathcal{D}$ *over associated parameter language* Lpar*, if there exist efficient simulators* $\mathsf{crssim}, \mathsf{sim}$ *such that for all non-uniform PPT adversaries* $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$ *we have (in all of the following probabilistic experiments, the experiment starts by setting* λ *as* $\lambda \leftarrow \mathsf{pargen}(1^m)$*, and choosing* ρ *as* $\rho \leftarrow \mathcal{D}_\lambda$*):*

Quasi-Adaptive Completeness:

$$\Pr\left[\begin{array}{c}\textsc{crs} \leftarrow \mathsf{crsgen}(\lambda, \rho); (x, w) \leftarrow \mathcal{A}_1(\textsc{crs}, \rho); \pi \leftarrow \mathsf{prover}(\textsc{crs}, x, w) : \\ \mathsf{ver}(\textsc{crs}, x, \pi) = 1 \ \textit{if}\ R_\rho(x, w)\end{array}\right] = 1$$

Quasi-Adaptive (Computational) Soundness:

$$\Pr[\textsc{crs} \leftarrow \mathsf{crsgen}(\lambda, \rho); (x, \pi) \leftarrow \mathcal{A}_2(\textsc{crs}, \rho) : x \notin L_\rho \ \wedge\ \mathsf{ver}(\textsc{crs}, x, \pi) = 1] \approx 0$$

Quasi-Adaptive Zero-Knowledge:

$$\Pr[\textsc{crs} \leftarrow \mathsf{crsgen}(\lambda, \rho) : \mathcal{A}_3^{\mathsf{prover}(\textsc{crs}, \cdot, \cdot)}(\textsc{crs}, \rho) = 1] \approx$$

$$\Pr[(\textsc{crs}, \mathsf{trap}) \leftarrow \mathsf{crssim}(\lambda, \rho) : \mathcal{A}_3^{\mathsf{sim}^*(\textsc{crs}, \mathsf{trap}, \cdot, \cdot)}(\textsc{crs}, \rho) = 1],$$

where $\mathsf{sim}^*(\textsc{crs}, \mathsf{trap}, x, w) = \mathsf{sim}(\textsc{crs}, \mathsf{trap}, x)$ *for* $(x, w) \in R_\rho$ *and both oracles (i.e.* prover *and* sim^**) output failure if* $(x, w) \notin R_\rho$*.*

We call a QA-NIZK **smooth (-verifier)** QA-NIZK if the verifier ver consists of three efficient algorithms $\mathsf{ver} = (\mathsf{hkgen}, \mathsf{pubH}, \mathsf{privH})$, and it satisfies the following modified completeness and soundness conditions. Here, hkgen is a probabilistic algorithm that takes a CRS as input and outputs two keys, hp, a projection hash key, and hk, a private hash key. The algorithm privH takes as input a word (e.g. a potential language member), and a (private hash) key, and outputs a string. Similarly, the algorithm pubH takes as input a proof (for instance generated by prover), and a (projection hash) key hp, and outputs a string.

The above **completeness** property is now defined as:

$$\Pr\left[\begin{array}{l}\textsc{crs} \leftarrow \mathsf{crsgen}(\lambda, \rho); (x, w) \leftarrow \mathcal{A}_1(\textsc{crs}, \rho); \pi \leftarrow \mathsf{prover}(\textsc{crs}, x, w); \\ (\mathsf{hp}, \mathsf{hk}) \leftarrow \mathsf{hkgen}(\textsc{crs}) : \ \ \mathsf{privH}(\mathsf{hk}, x) = \mathsf{pubH}(\mathsf{hp}, \pi) \text{ if } R_\rho(x, w)\end{array}\right] = 1$$

The QA-NIZK is said to satisfy **smooth-soundness** if for all words $x \notin L_\rho$, $\mathsf{privH}(\mathsf{hk}, x)$ is computationally indistinguishable to the Adversary from uniformly random, even when the Adversary is given hp, and even if it produces x after receiving hp.

More precisely, **Quasi-Adaptive Smooth-Soundness** is the following property (let $\mathcal{U}$ be the uniform distribution on the range of privH, which is assumed to be of cardinality exponential in m): for every two-stage efficient oracle adversary $\mathcal{A}$

$$\text{CRS} \leftarrow \mathsf{crsgen}(\lambda, \rho); (\mathsf{hp}, \mathsf{hk}) \leftarrow \mathsf{hkgen}(\text{CRS}); (x^*, \pi^*, \sigma) \leftarrow \mathcal{A}^{\mathcal{O}}(\text{CRS}, \rho, \mathsf{hp}); u \leftarrow \mathcal{U} :$$
$$\Pr[\mathcal{A}^{\mathcal{O}}(\mathsf{privH}(\mathsf{hk}, x^*), \sigma) = 1 \mid Q] - \Pr[\mathcal{A}^{\mathcal{O}}(u, \sigma) = 1 \mid Q] \approx 0$$

where the oracle $\mathcal{O}$ is instantiated with $\mathsf{privH}(\mathsf{hk}, \cdot)$, and Q is the condition that x^* is not in the language L_ρ **and** all oracle calls by the adversary in both stages are with L_ρ-language members. Here, σ is a local state of $\mathcal{A}$.

Note that as opposed to the information-theoretic smoothness property of projective hash functions, one cannot argue here that $\mathsf{privH}(\mathsf{hk}, x)$ for $x \in L_\rho$ can instead just be computed using hp, as that would also require efficiently computing a witness for x. Hence, the need to provide oracle access to $\mathsf{privH}(\mathsf{hk}, \cdot)$ for language members.

Also, note that smooth-soundness implies the earlier definition of soundness [JR13] if verification of (x, π) is defined as $\mathsf{privH}(\mathsf{hk}, x) = \mathsf{pubH}(\mathsf{hp}, \pi)$.

To differentiate the functionalities of the verifier of a QA-NIZK from similar functionalities of an SPHF, we will prepend the SPHF functionalities with keyword sphf and the QA-NIZK verifier functionalities with the keyword ver.

4 Matrix Decisional Assumptions

We will consider bilinear groups that consist of three cyclic groups of prime order q, $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ with an efficient bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Group elements $\mathbf{g}_1$ and $\mathbf{g}_2$ will typically denote generators of the group $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. Following [EHK$^+$13], in this section and the next we will use the notations $[a]_1, [a]_2$ and $[a]_T$ to denote $a\mathbf{g}_1, a\mathbf{g}_2$, and $a \cdot e(\mathbf{g}_1, \mathbf{g}_2)$ respectively and use additive notations for group operations. When talking about a general group $\mathbb{G}$ with generator $\mathbf{g}$, we will just use the notation $[a]$ to denote $a\mathbf{g}$. However, in the UC-PAKE constructions, we will switch to multiplicative notation for easy readability.

We recall the *Matrix Decisional Diffie Hellman* or MDDH assumptions from [EHK$^+$13]. A matrix distribution $\mathcal{D}_{l,k}$, where $l > k$, is defined to be an efficiently samplable distribution on $\mathbb{Z}_q^{l \times k}$ which is full-ranked with overwhelming probability. The *$\mathcal{D}_{l,k}$-MDDH assumption* in group $\mathbb{G}$ states that with samples $\mathbf{A} \leftarrow \mathcal{D}_{l,k}, \mathbf{s} \leftarrow \mathbb{Z}_q^k, \mathbf{s}' \leftarrow \mathbb{Z}_q^l$, the tuple $([\mathbf{A}], [\mathbf{A}\mathbf{s}])$ is computationally indistinguishable from $([\mathbf{A}], [\mathbf{s}'])$. A matrix distribution $\mathcal{D}_{k+1,k}$ is simply denoted by $\mathcal{D}_k$.

Intuitively, a $\mathcal{D}_{l,k}$-MDDH assumption allows us to generate l (computationally) independently random group elements from an initial k independently random exponents. A $\mathcal{D}_k$-MDDH assumption allows us to generate one extra random group element. In this section, we will establish that, in fact, a $\mathcal{D}_k$-MDDH assumption can be *boosted* to generate additional (computationally) independently random elements. This will be useful to us in the next section to prove the smoothness property of our construction.

We remark that boosting is different from the random self-reducibility of $\mathcal{D}_{l,k}$-MDDH assumptions, as described by [EHK+13]. While the former aims to generate extra randomness from the same initial sample of vector of random exponents, the latter talks about results from several independent samples of vector of random exponents. Boosting can be seen as an abstraction of the *switching lemma* of [JR14] and follows the same blueprint for the proof.

For an $l \times k$ matrix A, we denote $\bar{\mathsf{A}}$ to be the top $k \times k$ square sub-matrix of A and $\underline{\mathsf{A}}$ to be the bottom $(l-k) \times k$ sub-matrix of A.

Definition 2. *We say that a matrix distribution $\mathcal{D}_k$ on $\mathbb{Z}_q^{(k+1)\times k}$ is* boostable *to a matrix distribution $\mathcal{D}_{l,k}$ on $\mathbb{Z}_q^{l\times k}$, where $l > k$, if there are efficiently samplable distributions $\mathcal{E}$ on $\mathbb{Z}_q^{(l-k)\times k}$ and $\mathcal{F}$ on $\mathbb{Z}_q^{(l-k)\times(k+1)}$, such that the following hold:*

- *For $\mathsf{A} \leftarrow \mathcal{D}_k, \mathsf{B} \leftarrow \mathcal{D}_{l,k}, \mathsf{E} \leftarrow \mathcal{E}, \mathsf{F} \leftarrow \mathcal{F}$, we have:*

$$\bar{\mathsf{B}} \approx \bar{\mathsf{A}}, \quad \underline{\mathsf{B}} \approx \mathsf{E}\bar{\mathsf{A}} \approx \mathsf{F}\mathsf{A}.$$

- *For $\mathsf{F} \leftarrow \mathcal{F}$, with overwhelming probability, all entries of the rightmost column F_r of F are non-zero.*

Theorem 1. *If a matrix distribution $\mathcal{D}_k$ on $\mathbb{Z}_q^{(k+1)\times k}$ is boostable to a matrix distribution $\mathcal{D}_{l,k}$ on $\mathbb{Z}_q^{l\times k}$ then the $\mathcal{D}_k$-MDDH assumption implies the $\mathcal{D}_{l,k}$-MDDH assumption.*

Proof of this theorem can be found in Appendix A.

Corollary 1. *Any $\mathcal{D}_k$ distribution can be boosted to a $\mathcal{D}_{l,k}$ distribution which inherits the distribution of the top $k \times k$ matrix of the samples.*

This can be seen by setting the top $k \times k$ matrix of a $\mathcal{D}_{l,k}$ sample to be the top $k \times k$ matrix of a $\mathcal{D}_k$ sample and setting the bottom $(l-k)\times k$ sub-matrix of the $\mathcal{D}_{l,k}$ sample to be uniformly random in $\mathbb{Z}_q^{(l-k)\times k}$. The required distributions $\mathcal{E}$ and $\mathcal{F}$ are just the uniform distributions on their respective domains.

This corollary allows us to retain the *representation size* of the top square matrix of a $\mathcal{D}_k$ distribution sample, while boosting it to an assumption required for security proofs. In particular, in applications such as this paper, this can lead to shorter public keys.

5 Smooth Quasi-Adaptive NIZKs for Linear Subspaces

In this section we show that the single element QA-NIZK [JR14, KW15] for witness-samplable linear subspaces can easily be extended to be smooth QA-NIZK. Particularly, under SXDH, the public hash key hp generated by $\mathsf{ver.hkgen}$ consists of a single group element. Following [KW15], the result is proven under the more general MDDH assumption in bilinear groups.

We follow additive notation for group operations in this section. In later sections we will use product notation.

Linear Subspace Languages. We consider languages that are linear subspaces of vectors of $\mathbb{G}_1$ elements. In other words, the languages we are interested in can be characterized as languages parametrized by $[\mathbf{M}]_1$ as below:

$L_{[\mathbf{M}]_1} = \{[\mathbf{M}]_1\mathbf{x} \in \mathbb{G}_1^n \mid \mathbf{x} \in \mathbb{Z}_q^t\}$, where $[\mathbf{M}]_1$ is an $n \times t$ matrix of $\mathbb{G}_1$ elements.

Here $[\mathbf{M}]_1$ is an element of the associated *parameter language* Lpar, which is all $n \times t$ matrices of $\mathbb{G}_1$ elements. The parameter language Lpar also has a corresponding witness relation $\mathcal{R}_{\text{par}}$, where the witness is a matrix of $\mathbb{Z}_q$ elements : $\mathcal{R}_{\text{par}}([\mathbf{M}]_1, \mathbf{M})$ iff $[\mathbf{M}]_1 = \mathbf{M} \cdot \mathbf{g}_1$.

Robust and Efficiently Witness-Samplable Distributions. Let the $t \times n$ dimensional matrix $[\mathbf{M}]_1$ be chosen according to a distribution $\mathcal{D}$ on Lpar. The distribution $\mathcal{D}$ is called *robust* if with probability close to one the left-most t columns of $[\mathbf{M}]_1$ are full-ranked. A distribution $\mathcal{D}$ on Lpar is called *efficiently witness-samplable* if there is a probabilistic polynomial time algorithm such that it outputs a pair of matrices $([\mathbf{M}]_1, \mathbf{M})$ that satisfy the relation $\mathcal{R}_{\text{par}}$ (i.e., $\mathcal{R}_{\text{par}}([\mathbf{M}]_1, \mathbf{M})$ holds), and further the resulting distribution of the output $[\mathbf{M}]_1$ is same as $\mathcal{D}$. For example, the uniform distribution on Lpar is efficiently witness-samplable, by first picking $\mathbf{M}$ at random, and then computing $[\mathbf{M}]_1$.

Smooth QA-NIZK Construction. We now describe a smooth computationally-sound Quasi-Adaptive NIZK $(\mathsf{K}_0, \mathsf{K}_1, \mathsf{P}, \mathsf{V})$ for linear subspace languages $\{L_{[\mathbf{M}]_1}\}$ with parameters sampled from a robust and efficiently witness-samplable distribution $\mathcal{D}$ over the associated parameter language Lpar and given a $\mathcal{D}_k$-MDDH assumption.

Algorithm K_1: The algorithm K_1 generates the CRS as follows. Let $[\mathbf{M}^{n\times t}]_1$ be the parameter supplied to K_1. It generates an $n \times k$ matrix $\mathbf{K}$ with all elements chosen randomly from $\mathbb{Z}_q$ and a $(k+1)\times k$ matrix $\mathbf{A}$ from the MDDH distribution $\mathcal{D}_k$. Let $\bar{\mathbf{A}}$ be the top $k \times k$ square matrix of $\mathbf{A}$.

The common reference string (CRS) has two parts $\mathbf{CRS}_p$ and $\mathbf{CRS}_v$ which are to be used by the prover and the verifier respectively.

$$\mathbf{CRS}_p^{t\times k} := ([\mathbf{P}]_1 = [\mathbf{M}^\top \mathbf{K}]_1) \qquad \mathbf{CRS}_v := ([\mathbf{C}]_2^{n\times k} = [\mathbf{K}\bar{\mathbf{A}}]_2, \quad [\bar{\mathbf{A}}]_2^{k\times k})$$

Prover P**:** Given candidate $[\mathbf{y}]_1 = [\mathbf{M}]_1\mathbf{x}$ with witness vector $\mathbf{x}^{t\times 1}$, the prover generates the following proof consisting of k elements in $\mathbb{G}_1$:

$$\pi := \mathbf{x}^\top \mathbf{CRS}_p$$

Verifier V**:** The algorithm hkgen is as follows: Sample $\mathbf{s} \leftarrow \mathbb{Z}_q^k$. Given $\mathbf{CRS}_v$ as above, compute hk and hp as follows:

$$\mathsf{hk} := [\mathbf{C}]_2\ \mathbf{s}, \qquad \mathsf{hp} := [\bar{\mathbf{A}}]_2\ \mathbf{s}$$

The algorithms pubH and privH are as follows: Given candidate $[\mathbf{y}]_1$, and proof π, compute:

$$\mathsf{privH}(\mathsf{hk}, [\mathbf{y}]_1) := e([\mathbf{y}]_1^\top, \mathsf{hk}), \qquad \mathsf{pubH}(\mathsf{hp}, \pi) := e(\pi, \mathsf{hp})$$

Theorem 2. *The above algorithms* $(\mathsf{K}_0, \mathsf{K}_1, \mathsf{P}, \mathsf{V})$ *constitute a smooth computationally -sound Quasi-Adaptive NIZK proof system for linear subspace languages* $\{L_{[\mathbf{M}]_1}\}$ *with parameters* $[\mathbf{M}]_1$ *sampled from a robust and efficiently witness-samplable distribution* $\mathcal{D}$ *over the associated parameter language* Lpar*, given any group generation algorithm for which the* $\mathcal{D}_k$*-MDDH assumption holds for group* $\mathbb{G}_2$*.*

Proof. We now give a proof for smoothness. The proofs of completeness, zero knowledge and soundness are same as [KW15].

Smoothness: First, note that the range of privH is exponential in the security parameter, for otherwise an adversarial circuit can compute discrete logarithms with non-negligible probability. We prove smoothness by transforming the system over a sequence of games. Game $\mathbf{G}_0$ just replicates the construction, but samples $\mathbf{A}$ from a distribution $\mathcal{D}_{k+n-t,k}$ obtained by boosting the given distribution $\mathcal{D}_k$ by Corollary 1. The construction only uses the top $k \times k$ sub-matrix $\bar{\mathbf{A}}$ of the sample which is distributed identically for both $\mathcal{D}_k$ and $\mathcal{D}_{k+n-t,k}$. Let $\underline{\mathbf{A}}$ be the bottom $(n-t) \times k$ sub-matrix of $\mathbf{A}$.

In Game $\mathbf{G}_1$, the challenger efficiently samples $[\mathbf{M}]_1$ according to distribution $\mathcal{D}$, along with witness $\mathbf{M}$ (since $\mathcal{D}$ is an efficiently witness samplable distribution). Since $\mathbf{M}$ is an $n \times t$ dimensional rank t matrix, there is a rank $n-t$ matrix $\mathbf{M}^{\perp}$ of dimension $n \times (n-t)$ whose columns form a complete basis for the kernel of $\mathbf{M}^{\top}$, which means $\mathbf{M}^{\top}\mathbf{M}^{\perp} = \mathsf{0}^{t\times(n-t)}$. In this game, the NIZK CRS is computed as follows: Generate matrix $\mathbf{K}'$ uniformly randomly from $\mathbb{Z}_q^{n\times k}$ and compute the matrix $\mathbf{T}^{(n-t)\times k}$, such that $\mathbf{T}\bar{\mathbf{A}} = \underline{\mathbf{A}}$. Implicitly set: $\mathbf{K} = \mathbf{K}' + \mathbf{M}^{\perp}\mathbf{T}$. Therefore we have,

$$\mathbf{CRS}_p^{t\times k} = [\mathbf{M}^{\top}\mathbf{K}]_1 = [\mathbf{M}^{\top}(\mathbf{K}' + \mathbf{M}^{\perp}\mathbf{T})]_1 = [\mathbf{M}^{\top}\mathbf{K}']_1$$

$$[\mathbf{C}]_2^{n\times k} = [(\mathbf{K}' + \mathbf{M}^{\perp}\mathbf{T})\bar{\mathbf{A}}]_2 = \mathbf{K}'[\bar{\mathbf{A}}]_2 + \mathbf{M}^{\perp}[\underline{\mathbf{A}}]_2,$$

$$\mathsf{hk} = [\mathbf{C}]_2\ \mathbf{s}, \quad \mathsf{hp} = [\bar{\mathbf{A}}]_2\ \mathbf{s}$$

In Game $\mathbf{G}_2$, we sample a fresh random vector $\mathbf{s}'$ in $\mathbb{Z}_q^{n-t}$ and modify the simulated computations as follows:

$$\mathbf{CRS}_p^{t\times k} = [\mathbf{M}^{\top}\mathbf{K}']_1, \qquad [\mathbf{C}]_2^{n\times k} = \mathbf{K}'[\bar{\mathbf{A}}]_2 + \mathbf{M}^{\perp}[\underline{\mathbf{A}}]_2,$$

$$\mathsf{hk} = \mathbf{K}'[\bar{\mathbf{A}}\mathbf{s}]_2 + \mathbf{M}^{\perp}[\mathbf{s}']_2, \quad \mathsf{hp} = [\bar{\mathbf{A}}\mathbf{s}]_2$$

Given a $\mathcal{D}_{k+n-t,k}$ challenge which is either "real": $([\mathbf{A}]_2, [\bar{\mathbf{A}}\mathbf{s}]_2, [\underline{\mathbf{A}}\mathbf{s}]_2)$ or "fake": $([\mathbf{A}]_2, [\bar{\mathbf{A}}\mathbf{s}]_2, [\mathbf{s}']_2)$, we observe that the real tuple can be used to simulate Game $\mathbf{G}_1$, while the fake tuple can be used to simulate Game $\mathbf{G}_2$. Thus the games $\mathbf{G}_1$ and $\mathbf{G}_2$ are indistinguishable by the $\mathcal{D}_{k+n-t,k}$-MDDH assumption, which in turn is implied by the $\mathcal{D}_k$-MDDH assumption by Theorem 1.

Now in Game $\mathbf{G}_2$ we have,

$$\mathsf{privH}(\mathsf{hk}, [\mathbf{y}^*]_1) = e\left([\mathbf{y}^*]_1^{\top}, \mathbf{K}'[\bar{\mathbf{A}}\mathbf{s}]_2 + \mathbf{M}^{\perp}[\mathbf{s}']_2\right)$$

For the oracle queries where $[\mathbf{y}^*]_1 \in L_{[\mathsf{M}]_1}$, we have $\mathbf{y}^{*\top}\mathsf{M}^{\perp} = 0^{1\times(n-t)}$. Hence the simulator responds with $e\left([\mathbf{y}^*]_1^{\top}, \mathsf{K}'[\bar{\mathsf{A}}\mathbf{s}]_2\right)$. Note that $\mathbf{s}'$ does not appear in this response.

For the adversary supplied $[\mathbf{y}^*]_1 \notin L_{[\mathsf{M}]_1}$, we have $\mathbf{y}^{*\top}\mathsf{M}^{\perp} \neq 0^{1\times(n-t)}$. Therefore $\mathsf{privH}(\mathsf{hk}, \mathbf{y}^*)$ is uniformly random, as $\mathbf{s}'$ is independently random of everything else given to the adversary. □

6 Asymmetric UC-PAKE: UC-APAKE

6.1 The UC Ideal Functionality for Asymmetric PAKE

Based on the UC-PAKE functionality of [CHK$^+$05], Gentry et al [GMR06] gave another UC functionality for asymmetric PAKE (UC-APAKE). A salient feature of the UC-PAKE functionality [CHK$^+$05] is that it models the security requirement that an adversary cannot perform efficient off-line computations on protocol transcripts to verifiably guess the low-entropy password. An adversary can only benefit from the low-entropy of the password by actually conducting an on-line attack (i.e. by impersonating one of the parties with a guessed password). This is modeled in the ideal world with a TestPwd capability available to the ideal world adversary: if TestPwd is called with the correct password, the ideal world adversary is allowed to set the session key. Moreover, in this functionality if any of the parties is corrupted, then the ideal world adversary is given the registered password.

In asymmetric PAKE [GMR06], the ideal functionality also allows an adversary to steal the password file stored at the server (while not necessarily corrupting the server). However, this by itself does not directly provide the actual password to the adversary. However, after this point the adversary is allowed to perform OfflineTestPwd tests to mimic a similar capability in the real world (in fact, the ideal world adversary is even allowed to perform OfflineTestPwd tests before it steals the password file, but it does not get a confirmation of the guess being correct until after it steals the password file).

Moreover, after the "steal password file" event the adversary is also allowed to impersonate the server to a *correctly guessed* client, even without providing the actual password (as it can clearly do so in the real world). However, compromising impersonation of the client still requires providing a correct password. This differentiation in capabilities also becomes important when characterizing the complexity of a simulator in terms of the real world adversary, as we will see later.

The $\mathcal{F}_{\text{PAKE}}$ functionality for UC-PAKE was a single-session functionality. However, asymmetric PAKE requires that a password file be used across multiple sessions, so the $\mathcal{F}_{\mathsf{apwKE}}$ functionality for UC-APAKE is defined as a multiple-session functionality. Note that this cannot be accomplished simply using composition with joint state [CR03] because the functionality itself requires shared state that needs to be maintained between sessions.

The complete UC-APAKE functionality $\mathcal{F}_{\mathsf{apwKE}}$ is described in detail in Fig. 1.

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

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

Password Storage and Authentication Sessions

Upon receiving a query $(\mathsf{StorePwdFile}, \boldsymbol{sid}, P_i, pw)$ **from party** P_j**:**
If this is the first $\mathsf{StorePwdFile}$ query, store password data record $(\mathsf{file}, P_i, P_j, pw)$ and mark it $\mathsf{uncompromised}$.

Upon receiving a query $(\mathsf{CltSession}, \boldsymbol{sid}, \mathsf{ssid}, P_i, P_j, pw)$ **from party** P_i**:**
Send $(\mathsf{CltSession}, sid, \mathsf{ssid}, P_i, P_j)$ to S. In addition, if this is the first $\mathsf{CltSession}$ query for ssid, then store session record $(\mathsf{Clt}, \mathsf{ssid}, P_i, P_j, pw)$ and mark this record fresh.

Upon receiving a query $(\mathsf{SvrSession}, \boldsymbol{sid}, \mathsf{ssid})$ **from party** P_j**:**
If there is a password data record $(\mathsf{file}, P_i, P_j, pw)$, then send $(\mathsf{SvrSession}, sid, \mathsf{ssid}, P_j, P_i)$ to S, and if this is the first $\mathsf{SvrSession}$ query for ssid, store session record $(\mathsf{Svr}, \mathsf{ssid}, P_j, P_i, pw)$, and mark it fresh.

Stealing Password Data

Upon receiving a query $(\mathsf{StealPwdFile}, \boldsymbol{sid})$ **from adversary** S**:**
If there is no password data record reply to S with 'no password file'. Otherwise, do the folloing: If the password data record $(\mathsf{file}, P_i, P_j, pw)$ is marked $\mathsf{uncompromised}$, mark it $\mathsf{compromised}$. If there is a tuple $(\mathsf{offline}, pw')$ stored with $pw' = pw$ then send pw to S, otherwise reply to S with 'password file stolen'.

Upon receiving a query $(\mathsf{OfflineTestPwd}, \boldsymbol{sid}, pw')$ **from Adversary** S**:**
If there is no password data record, or if there is a password data record $(\mathsf{file}, P_i, P_j, pw)$ that is marked $\mathsf{uncompromised}$, then store $(\mathsf{offline}, pw')$. Otherwise do: if $pw = pw'$, send pw back to S. If $pw \neq pw'$, reply with 'wrong guess'.

Active Session Attacks

Upon receiving a query $(\mathsf{TestPwd}, \boldsymbol{sid}, \mathsf{ssid}, P_i, pw')$ **from the adversary** S**:**
If there is a session record of the form $(\mathsf{role}, \mathsf{ssid}, P_i, P_j, pw)$ which is fresh, then do: If $pw = pw'$, mark the record $\mathsf{compromised}$ and reply to S with "correct guess". If $pw \neq pw'$, mark the record $\mathsf{interrupted}$ and reply with "wrong guess".

Upon receiving a query $(\mathsf{Impersonate}, \boldsymbol{sid}, \mathsf{ssid})$
If there is a session record of the form $(\mathsf{Clt}, \mathsf{ssid}, P_i, P_j, pw)$ which is fresh, then do: then if there is a password data record file $(\mathsf{file}, P_i, P_j, pw)$ that is marked $\mathsf{compromised}$, mark the session record $\mathsf{compromised}$ and reply to S with 'correct guess', else mark the session record $\mathsf{interrupted}$ and reply with wrong guess.

Key Generation and Authentication

Upon receiving a query $(\mathsf{NewKey}, \boldsymbol{sid}, \mathsf{ssid}, P_i, sk)$ **from** S**, where** $|sk| = k$**:**
If there is a session record of the form $(\mathsf{role}, \mathsf{ssid}, P_i, P_j, pw)$ that is not marked $\mathsf{completed}$,

– If this record is $\mathsf{compromised}$, or either P_i or P_j is corrupted, then output (sid, ssid, sk) to player P_i.
– If this record is fresh, and there is a session record $(\mathsf{role}, \mathsf{ssid}, P_j, P_i, pw')$ with $pw' = pw$, and a key sk' was sent to P_j, and $(\mathsf{role}, \mathsf{ssid}, P_j, P_i, pw)$ was fresh at the time, then output $(sid, \mathsf{ssid}, sk')$ to P_i.
– In any other case, pick a new random key sk' of length k and send $(sid, \mathsf{ssid}, sk')$ to P_i.

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

Upon receiving $(\mathsf{Corrupt}, \boldsymbol{sid}, P)$ **from** S**:** if there is a $(\mathsf{Clt}, sid, P, P', pw)$ recorded, return pw to S, and mark P_i corrupted. If there is a $(\mathsf{Svr}, sid, P, P', pw)$ recorded, then mark P corrupted and (internally) call $(\mathsf{StealPwdFile}, sid)$.

Fig. 1. The password-based key-exchange functionality $\mathcal{F}_{\mathsf{apwKE}}$

6.2 UC-APAKE based on VPAKE and Smooth-NIZK

We now design an asymmetric UC-PAKE based on Verifier-based PAKE or VPAKE of Benhamooda and Pointcheval [BP13] and the novel Smooth NIZK proofs. The essential idea of [BP13] is that while the Client holds the actual

Generate $\mathbf{g} \leftarrow \mathbb{G}_1$; $a_1, a_2, b_{\mathsf{C}}, b_{\mathsf{S}} \leftarrow \mathbb{Z}_q$ and let $\rho = \{\mathbf{a}_1 = \mathbf{g}^{a_1}, \mathbf{a}_2 = \mathbf{g}^{a_2}, \mathbf{b}_{\mathsf{C}} = \mathbf{g}^{b_{\mathsf{C}}}, \mathbf{b}_{\mathsf{S}} = \mathbf{g}^{b_{\mathsf{S}}}\}$.

Define languages $\begin{bmatrix} L_{\mathsf{C}} = \{(R,S,H) \mid \exists r,p : R = \mathbf{g}^r, S = \mathbf{a}_1^r \mathbf{b}_{\mathsf{C}}^p, H = \mathbf{b}_{\mathsf{S}}^p\} \\ L_{\mathsf{S}} = \{(R,S) \mid \exists r : R = \mathbf{g}^r, S = \mathbf{a}_2^r\} \end{bmatrix}$

Let $(\mathsf{hk}_{\mathsf{C}}, \mathsf{hp}_{\mathsf{C}}) \leftarrow \mathsf{sphf}(L_{\mathsf{C}}).\mathsf{hkgen}$ and $(\mathsf{hp}_{\mathsf{S}}, \mathsf{hk}_{\mathsf{S}}) \leftarrow \mathsf{sphf}(L_{\mathsf{S}}).\mathsf{hkgen}$.

Define languages:

$$\begin{bmatrix} L_{\mathsf{C}}^{+} = \{(R,S,H,T,l) \mid \exists r,p : R = \mathbf{g}^r, S = \mathbf{a}_1^r \mathbf{b}_{\mathsf{C}}^p, H = \mathbf{b}_{\mathsf{S}}^p, T = \mathsf{sphf.pubH}(\mathsf{hp}_{\mathsf{C}}, \langle R,S,H\rangle, l; r, p)\} \\ L_{\mathsf{S}}^{+} = \{(R,S,T,l) \mid \exists r : R = \mathbf{g}^r, S = \mathbf{a}_2^r, T = \mathsf{sphf.pubH}(\mathsf{hp}_{\mathsf{S}}, \langle R,S\rangle, l; r)\} \end{bmatrix}$$

Let $(\mathsf{pargen}_P, \mathsf{crsgen}_P, \mathsf{prover}_P, \mathsf{ver}_P)$ be Smooth QA-NIZKs for languages L_P^{+}, with $P \in \{C, S\}$.

Let $\textsc{crs}_P \leftarrow \mathsf{crsgen}_P(\rho)$ and $\mathcal{H}$ be a collision resistant hash function.

Let $\mathcal{RO}$ be a random oracle and let $\mathrm{phash} = \mathcal{RO}(sid, P_i, P_j, \mathrm{pwd})$.

$$\mathrm{CRS} := (\rho, \mathsf{hp}_{\mathsf{C}}, \mathsf{hp}_{\mathsf{S}}, \textsc{crs}_{\mathsf{C}}, \textsc{crs}_{\mathsf{S}}, \mathcal{H}).$$

$$\text{Server Persistent State} := \mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}.$$

Client P_i	Network
Input $(\mathsf{CltSession}, sid, \mathsf{ssid}, P_i, P_j, \mathrm{pwd})$. Choose $r_1 \leftarrow \mathbb{Z}_q$ and $(\textsc{hk}_1, \textsc{hp}_1) \leftarrow \mathsf{ver}_{\mathsf{S}}.\mathsf{hkgen}(\textsc{crs}_{\mathsf{S}})$. Set $R_1 = \mathbf{g}^{r_1}$, $S_1 = \mathbf{a}_1^{r_1} \mathbf{b}_{\mathsf{C}}^{\mathrm{phash}}$, $T_1 = \mathsf{sphf}_{\mathsf{C}}.\mathsf{pubH}(\mathsf{hp}_{\mathsf{C}}, \langle R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}\rangle, i_1; r_1, \mathrm{phash})$, $W_1 = \mathsf{prover}_{\mathsf{C}}(\textsc{crs}_{\mathsf{C}}, \langle R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}, T_1, i_1\rangle; r_1, \mathrm{phash})$, where $i_1 = \mathcal{H}(sid, \mathsf{ssid}, P_i, P_j, R_1, S_1, \textsc{hp}_1)$. Erase r_1, send $(R_1, S_1, T_1, \textsc{hp}_1)$ and retain $(W_1, \textsc{hk}_1)$.	$\xrightarrow{R_1, S_1, T_1, \mathrm{HP}_1} P_j$
Receive $(R_2', S_2', T_2', \textsc{hp}_2')$. If any of $R_2', S_2', T_2', \textsc{hp}_2'$ is not in their respective group or is 1, set $\mathrm{sk}_1 \overset{\$}{\leftarrow} \mathbb{G}_T$, else compute $i_2' = \mathcal{H}(sid, \mathsf{ssid}, P_j, P_i, R_2', S_2', \textsc{hp}_2')$, and $\mathrm{sk}_1 = \mathsf{ver}_{\mathsf{S}}.\mathsf{privH}(\textsc{hk}_1, \langle R_2', S_2'/\mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}, T_2', i_2'\rangle) \cdot \mathsf{ver}_{\mathsf{C}}.\mathsf{pubH}(\textsc{hp}_2', W_1)$. Output $(sid, \mathsf{ssid}, \mathrm{sk}_1)$.	$\xleftarrow{R_2', S_2', T_2', \mathrm{HP}_2'} P_j$

Server P_j	Network
Input $(\mathsf{SvrSession}, sid, \mathsf{ssid}, P_j, P_i, \text{Server Persistent State})$. Choose $r_2 \leftarrow \mathbb{Z}_q$ and $(\textsc{hk}_2, \textsc{hp}_2) \leftarrow \mathsf{ver}_{\mathsf{C}}.\mathsf{hkgen}(\textsc{crs}_{\mathsf{C}})$. Set $R_2 = \mathbf{g}^{r_2}$, $S_2 = \mathbf{a}_2^{r_2} \mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}$, $T_2 = \mathsf{sphf}_{\mathsf{S}}.\mathsf{pubH}(\mathsf{hp}_{\mathsf{S}}, \langle R_2, S_2/\mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}\rangle, i_2; r_2)$, $W_2 = \mathsf{prover}_{\mathsf{S}}(\textsc{crs}_{\mathsf{S}}, \langle R_2, S_2/\mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}, T_2, i_2\rangle; r_2)$, where $i_2 = \mathcal{H}(sid, \mathsf{ssid}, P_j, P_i, R_2, S_2, \textsc{hp}_2)$. Erase r_2, send $(R_2, S_2, T_2, \textsc{hp}_2)$ and retain $(W_2, \textsc{hk}_2)$.	$\xrightarrow{R_2, S_2, T_2, \mathrm{HP}_2} P_i$
Receive $(R_1', S_1', T_1', \textsc{hp}_1')$. If any of $R_1', S_1', T_1', \textsc{hp}_1'$ is not in their respective group or is 1, set $\mathrm{sk}_2 \overset{\$}{\leftarrow} \mathbb{G}_T$, else compute $i_1' = \mathcal{H}(sid, \mathsf{ssid}, P_i, P_j, R_1', S_1', \textsc{hp}_1')$, and $\mathrm{sk}_2 = \mathsf{ver}_{\mathsf{C}}.\mathsf{privH}(\textsc{hk}_2, \langle R_1', S_1', \mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}, T_1', i_1'\rangle) \cdot \mathsf{ver}_{\mathsf{S}}.\mathsf{pubH}(\textsc{hp}_1', W_2)$. Output $(sid, \mathsf{ssid}, \mathrm{sk}_2)$.	$\xleftarrow{R_1', S_1', T_1', \mathrm{HP}_1'} P_i$

Fig. 2. Single round RO-hybrid UC-APAKE protocol under SXDH assumption.

password, the Server does not hold password in the clear. Instead the Server stores a hard to invert function called PHash (password hash) evaluated over the password and a random "salt" (PSalt) published in the CRS. While executing a session, the client sends encryptions of the password or another function called PPreHash (password pre-hash) evaluated on the password. Correspondingly, the server sends encryptions of the stored PHash.

Of course, some kind of zero-knowledge proof must accompany these encryptions, and to that end [BP13] can utilize the new smooth projective hash functions (SPHF) for CCA2-encryption [BBC$^+$13] such as Cramer-Shoup encryption [CS02]. In each session, both parties generate fresh SPHF private and projection keys (to be employed on incoming messages). The projection key is sent (piggy-backed) along with the encrypted message. If the encrypted messages use the correct password (meaning both parties have the same password or its PHash), then SPHF computed on the message by the receiving party using the SPHF hash key it generated equals the SPHF computed on the message by the sending party using the SPHF projection key it received. Thus, these SPHF hashes can be used to compute the session key. Smoothness property of the SPHF guarantees security of the VPAKE scheme.

Unfortunately, each party must retain the witness used in the CCA2 encryption, as computing the SPHF projection-hash of its outgoing encrypted message using the received projection key requires this witness. In the strong simulation paradigm of universally composable security, this leads to a problem if an Adversary can corrupt a session dynamically after the outgoing message has been sent and the incoming message has not yet been received. Thus, this SPHF methodology can only handle static corruption. While Jutla and Roy [JR15] have recently given an efficient UC-PAKE protocol which can handle dynamic corruption, the construction uses ideas from dual-system simulation-sound QA-NIZK that they introduce there. These ideas are rather intricate and do not seem to allow a modular or generic design of such UC password-authenticated protocols.

In this paper, we show that the new notion of Smooth QA-NIZK allows easy to understand (and equally efficient) modular or generic design. Just as QA-NIZK proofs can be seen as generalization of projective hash proof systems to public verifiability (and also assuring zero-knowledge), the novel notion of Smooth QA-NIZK naturally generalizes the notion of smooth projective hash functions where instead of the witness, the publicly verifiable proof can be used to evaluate the projection-hash. The zero-knowledge property of this publicly verifiable proof assures that this proof and hence the projection-hash can be generated by a simulator with no access to the witness. In particular, each party in the UC-PAKE protocol can generate an encryption of the password and generate this publicly verifiable QA-NIZK proof, send the encryption to the other party, erase the witness and retain just the proof for later generation of session key.

The natural question that arises is whether one needs a notion of smooth-soundness under simulation. Indeed, one does need some form of unbounded simulation-soundness as the UC simulator generates QA-NIZK proofs on non-language members without access to the password. Unfortunately, the recent efficient unbounded simulation sound QA-NIZK construction of [KW15] does not extend to be smooth under unbounded simulation (or at least current techniques do not seem to allow one to prove so). The dual-system simulation sound QA-NIZK [JR15] does satisfy smoothness property, but it would need introduction of various new intricate definitions and complicated proofs. One may also ask whether CCA2 encryption by itself provides the required simulation soundness, but that is also not the case, as CCA2 encryption by itself does not give a privately-verifiable (say, via its underlying SPHF as in Cramer-Shoup encryption) proof that it is the password that is being encrypted.

In light of this, it turns out that the simplest way to design the UC-APAKE (or UC-PAKE) protocol is to use an El-Gamal encryption of the password (or its $\mathsf{PPreHash}$ or PHash) and augment it with an SPHF proof of its consistency, and finally a Smooth QA-NIZK on this augmented El-Gamal encryption. (If the reader is interested in the simpler UC-PAKE protocol secure under dynamic corruption in the new Smooth QA-NIZK framework, the protocol and proof are provided in the Appendix.)

We will also need the random oracle hybrid model to achieve the goal of a UC-APAKE protocol, as explained next. The focus of [BP13] was to design protocols which can be proven secure in the standard model. They formalized a security notion for APAKEs modifying the game-based BPR model [BPR00]. However, our focus is to construct an APAKE protocol in the UC model. In the UC model of [GMR06], the UC simulator must be able to detect offline password guess attempts of the adversary. This is not possible in the standard model as offline tests can be internally performed by the adversary. In order to intercept offline tests by the adversary, it thus becomes inevitable to use an idealized model, such as the random oracle model.

So in particular, we adapt the random oracle-based password hashing scheme of [BP13]. In the scheme, the public parameters are $param = \mathbf{b}_{\mathsf{C}}, \mathbf{b}_{\mathsf{S}}$ randomly sampled from $\mathbb{G}_1$ and a random oracle $\mathcal{RO}$. Define phash $= \mathcal{RO}(sid,$ Client-id, Server-id, pwd), where Client-id, Server-id are the ids of the participating parties, *sid* is the common session-id for all sessions between these parties and pwd is the password of the client. We set:

$$\begin{aligned}&\mathsf{PPreHash}(param, \mathrm{pwd}) = \mathbf{b}_{\mathsf{C}}^{\mathrm{phash}}\\ &\mathsf{PSalt}(param) = \mathbf{b}_{\mathsf{S}}\\ &\mathsf{PHash}(param, \mathrm{pwd}) = \mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}\end{aligned}$$

Corresponding to the asymmetric storages of the client and the server, we define the following languages, one for each party, which implicitly check the consistency of correct elements being used:

$$\begin{aligned}&L_{\mathsf{C}} = \{(R, S, H) \mid \exists r, p : R = \mathbf{g}^r, S = \mathbf{a}_1^r \mathbf{b}_{\mathsf{C}}^p, H = \mathbf{b}_{\mathsf{S}}^p\}\\ &L_{\mathsf{S}} = \{(R, S) \mid \exists r : R = \mathbf{g}^r, S = \mathbf{a}_2^r\}\end{aligned}$$

We now plug these languages into UC-PAKE methodology described above. The client sends ElGamal encryption of $\mathbf{b}_{\mathsf{C}}^p$, as in (R, S) of L_{C}, while the server supplies the last element H for forming a word of L_{C}. The server sends ElGamal encryption of $\mathbf{b}_{\mathsf{S}}^{p}$, while the client divides out $\mathbf{b}_{\mathsf{S}}^{p}$ from the second component to form a word of L_{S}.

The CRS provides public smooth$_2$ SPHF keys for the languages L_{C} and L_{S}, which are used by the client and server respectively to compute T_1 and T_2 for their flows.

Lastly, we use Smooth QA-NIZK proofs for generating a public hash key and a private hash key over the above languages augmented with the SPHFs as below:

$$L_{\mathsf{C}}^{+} = \left\{ \begin{array}{c} (R, S, H, T, l) \mid \exists r, p : R = \mathbf{g}^{r}, S = \mathbf{a}_1^{r}\mathbf{b}_{\mathsf{C}}^{p}, H = \mathbf{b}_{\mathsf{S}}^{p}, \\ T = \mathsf{sphf.pubH}(\mathsf{hp}_{\mathsf{C}}, \langle R, S, H \rangle, l; r, p) \end{array} \right\}$$
$$L_{\mathsf{S}}^{+} = \{(R, S, T, l) \mid \exists r : R = \mathbf{g}^{r}, S = \mathbf{a}_2^{r}, T = \mathsf{sphf.pubH}(\mathsf{hp}_{\mathsf{S}}, \langle R, S \rangle, l; r)\}$$

The client generates a Smooth QA-NIZK verification key pair for the server language L_{S}^{+}, retains the private key $\textsc{hk}_1$ and sends the public key $\textsc{hp}_1$ along with the ElGamal encryption and the SPHF. The client computes a QA-NIZK proof W_1 of $(R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\text{phash}}, T_1) \in L_{\mathsf{C}}^{+}$ with label $i_1 = \mathcal{H}(\mathit{sid}, \mathsf{ssid}, P_i, P_j, R_1, S_1, T_1, \textsc{hp}_1)$ and retains that for later key computation.

Similarly, the server generates a Smooth QA-NIZK verification key pair for the client language L_{C}^{+}, retains the private key $\textsc{hk}_2$ and sends the public key $\textsc{hp}_2$ along with the ElGamal encryption and the SPHF. The server computes a QA-NIZK proof W_2 of $(R_2, S_2/\mathbf{b}_{\mathsf{S}}^{\text{phash}}, T_2) \in L_{\mathsf{S}}^{+}$ with label $i_2 = \mathcal{H}(\mathit{sid}, \mathsf{ssid}, P_j, P_i, R_2, S_2, T_2, \textsc{hp}_2)$ and retains that for later key computation.

In the second part of the protocol, after receiving the peer flow, each party computes the final secret key as the product of the private Smooth QA-NIZK hash of the peer flow with own private Smooth QA-NIZK key and the public Smooth QA-NIZK hash of the (retained) QA-NIZK proof of own flow with the peer public Smooth QA-NIZK hash key. Formally the client computes:

$$\mathsf{ver}_{\mathsf{S}}.\mathsf{privH}(\textsc{hk}_1, \langle R_2', S_2'/\mathbf{b}_{\mathsf{S}}^{\text{phash}}, T_2', i_2' \rangle) \cdot \mathsf{ver}_{\mathsf{C}}.\mathsf{pubH}(\textsc{hp}_2', W_1).$$

Similarly, the server computes:

$$\mathsf{ver}_{\mathsf{C}}.\mathsf{privH}(\textsc{hk}_2, \langle R_1', S_1', \mathbf{b}_{\mathsf{S}}^{\text{phash}}, T_1', i_1' \rangle) \cdot \mathsf{ver}_{\mathsf{S}}.\mathsf{pubH}(\textsc{hp}_1', W_2).$$

Given the completeness property of the Smooth QA-NIZK, it is not difficult to see that legitimately completed peer sessions end up with equal keys. In the next section, we prove that this protocol securely realizes $\mathcal{F}_{\mathsf{apwKE}}$, as stated in the theorem below.

The complete protocol is described in detail in Figure 2. The SPHF sphf is required to be a smooth$_2$ projective hash function (see Section 2 for definitions). For simplicity, in this paper we focus on constructions based on $\mathcal{D}_1$-MDDH assumptions, and in particular the SXDH assumption (see Appendix B).

Theorem 3. *Under the $\mathcal{D}_1$-MDDH assumption SXDH, the protocol in Fig. 2 securely realizes the $\mathcal{F}_{\mathsf{apwKE}}$ functionality in the $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{RO}})$-hybrid model, in the presence of adaptive corruption adversaries. The number of unique password* *arguments passed to* TestPwd *and* OfflineTestPwd *of* $\mathcal{F}_{\mathsf{apwKE}}$ *combined in the ideal world is at most the number of random oracle calls in the* $(\mathcal{F}_{\text{CRS}}, \mathcal{F}_{\text{RO}})$*-hybrid world.*

6.3 Main Idea of the UC Simulator

The UC simulator $\mathcal{S}$ works as follows: It simulates the random oracle calls and records all the query response pairs. It will generate the CRS for $\widehat{\mathcal{F}}_{\text{PAKE}}$ using the real world algorithms, except for the Smooth QA-NIZK, for which it uses the simulated CRS generator. It also retains the private hash keys of the SPHF's. The next main difference is in the simulation of the outgoing message of the real world parties: $\mathcal{S}$ uses a dummy message μ instead of the real password which it does not have access to. Further, it postpones computation of W till the session-key generation time. Finally, another difference is in the processing of the incoming message, where $\mathcal{S}$ decrypts the incoming message R_2', S_2' and runs through the list of random oracle queries to search for a pwd$'$, such that the decryption is $\mathbf{b}_{\mathsf{S}}^{\mathcal{RO}(sid, P_i, P_j, \text{pwd}')}$, which it uses to call the ideal functionality's test function. It next generates an sk similar to how it is generated in the real-world. It sends sk to the ideal functionality to be output to the party concerned.

Since the (R_1, S_1) that it sends out is no longer such that $(R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\text{phash}})$ in the language L_{C}, it has to use the private key of the SPHF in order to compute T_1 on $(R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\text{phash}})$ and the QA-NIZK proof simulator to compute W_1.

There are other special steps designed to simulate stealing the password file and then impersonating the server to the client. Specifically, when the password file is stolen, the simulator still may not know pwd. It then preemptively sets phash to a random value and pretends that this is the random oracle response with the correct pwd query. Later on when there is a successful pwd query, which the simulator can find out by the online or offline testpwd ideal functionality calls, it sets the record accordingly.

In case of a stolen password file, the simulator includes a "Client Only Step" which lets it test (modified) server flows for consistency and call the Impersonate functionality if consistency checks out. The server simulation steps do not include such a step to model the security notion that even if the password file is stolen, the adversary should still not be able to impersonate the client.

6.4 Main Idea of the Proof of UC Realization

The proof that the simulator $\mathcal{S}$ described above simulates the Adversary in the real-world protocol, follows essentially from the properties of the Smooth QA-NIZK and smooth$_2$ SPHF, and we give a broad outline here. The proof will describe various experiments between a challenger $\mathcal{C}$ and the adversary, which we will just assume to be the environment $\mathcal{Z}$ (as the adversary $\mathcal{A}$ can be assumed to be just dummy and following $\mathcal{Z}$'s commands). In the first experiment the challenger $\mathcal{C}$ will just be the combination of the code of the simulator $\mathcal{S}$ above and $\widehat{\mathcal{F}}_{\text{PAKE}}$. In particular, after the environment issues a CltSession request with a password pwd, the challenger gets that password. So, while in the first experiment, the challenger (copying $\mathcal{S}$) does not use pwd directly, from the next experiment onwards, it can use pwd. Thus, the main goal of the ensuing experiments is to modify the fake tuples $\mathbf{g}^{r_1}, \mathbf{a}_1^{r_1} \mathbf{b}_{\mathsf{C}}^{\mu} \cdot \mathbf{g}^{r'}$ by real tuples (as in real-world) $\mathbf{g}^{r_1}, \mathbf{a}_1^{r_1} \mathbf{b}_{\mathsf{C}}^{\mathrm{phash}}$, since the challenger has access to pwd, and hence phash. This is accomplished by a hybrid argument, modifying one instance at a time using DDH assumption in group $\mathbb{G}_1$.

The guarantee that the client cannot be impersonated by the adversary, even when the password file is stolen is established by noting that $\mathbf{b}_{\mathsf{C}}^{\mathrm{phash}}$, which is what the client encrypts in its flows, is hard to compute given the server persistent state $\mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}$. This is formally captured in the proof by using a DDH transition from $(\mathbf{b}_{\mathsf{S}}, \mathbf{b}_{\mathsf{C}}, \mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}, \mathbf{b}_{\mathsf{C}}^{\mathrm{phash}})$ to $(\mathbf{b}_{\mathsf{S}}, \mathbf{b}_{\mathsf{C}}, \mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}, \mathbf{b}_{\mathsf{C}}^{z})$, where z is independently random from phash.

Once all the instances are corrected, i.e. R_1, S_1 are generated as $\mathbf{g}^{r_1}, \mathbf{a}_1^{r_1} \mathbf{b}_{\mathsf{C}}^{\mathrm{phash}}$, the challenger can switch to the real-world because the tuples $R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}$ are now in the language L_{C}. This implies that the session keys are generated exactly as in the real-world.

6.5 Adaptive Corruption

The UC protocol described above is also UC-secure against adaptive corruption of parties by the Adversary in the erasure model. In the real-world when the adversary corrupts a client (with a Corrupt command), it gets the internal state of the client. Clearly, if the party has already been invoked with a CltSession command then the password pwd is leaked at the minimum, and hence the ideal functionality $\mathcal{F}_{\text{PAKE}}$ leaks the password to the Adversary in the ideal world. In the protocol described above, the Adversary also gets W_1 and HK_1, as this is the only state maintained by each client between sending $R_1, S_1, T_1, \mathrm{HP}_1$, and the final issuance of session-key. Simulation of HK_1 is easy for the simulator $\mathcal{S}$ since $\mathcal{S}$ generates HK_1 exactly as in the real world. For generating W_1, which $\mathcal{S}$ had postponed to computing till it received an incoming message from the adversary, it can now use the pwd which it gets from $\widehat{\mathcal{F}}_{\text{PAKE}}$ by issuing a Corrupt call to $\widehat{\mathcal{F}}_{\text{PAKE}}$. More precisely, it issues the Corrupt call, and gets pwd, and then calls the QA-NIZK simulator with the tuple $(R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}, T_1, i_1)$ to get W_1. Note that this computation of W_1 is identical to the postponed computation of W_1 in the computation of client factor of sk_1 (which is really used in the output to the environment when $\mathrm{pwd}' = \mathrm{pwd}$).

In case of server corruption, the simulator does not get pwd, but is able to set phash which also enables it to compute W_2 using the QA-NIZK simulator on $(R_2, S_2/\mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}, T_2, i_2)$.

6.6 Simulator for the Protocol

We will assume that the adversary $\mathcal{A}$ in the UC protocol is dummy, and essentially passes back and forth commands and messages from the environment $\mathcal{Z}$.

Thus, from now on we will use environment $\mathcal{Z}$ as the real adversary, which outputs a single bit. The simulator $\mathcal{S}$ will be the ideal world adversary for $\mathcal{F}_{\mathsf{apwKE}}$. It is a universal simulator that uses $\mathcal{A}$ as a black box. For each instance (session and a party), we will use a prime, to refer to variables received in the message from $\mathcal{Z}$ (i.e. $\mathcal{A}$). We will call a message *legitimate* if it was not altered by $\mathcal{Z}$, and delivered in the correct session and to the correct party.

Responding to random oracle queries. Let the input be m. If there is a record of the form (m, r), that is, m was queried before and was responded with r, then just return r.

Otherwise, if m is of the form (sid, P_i, P_j, x), for some x and the password file has been stolen then call $\mathsf{OfflineTestPwd}$ with x. If the test succeeds then return phash, which must already have been set (see Stealing Password File below), and record (m, phash).

In all other cases, generate $r \leftarrow \mathbb{Z}_q$, record (m, r) and return r.

Setting the CRS. The simulator $\mathcal{S}$ picks the CRS just as in the real world, except the QA-NIZK CRS-es are generated using the crs-simulators, which also generate simulator trapdoors $\tau_{\mathsf{C}}, \tau_{\mathsf{S}}$. It retains $a_1, a_2, \tau_{\mathsf{C}}, \tau_{\mathsf{S}}, \mathsf{hk}_{\mathsf{C}}, \mathsf{hk}_{\mathsf{S}}$ as trapdoors.

New Client Session: Sending a message to $\mathcal{Z}$. On message ($\mathsf{CltSession}$, *sid*, ssid, P_i, P_j) from $\mathcal{F}_{\mathsf{apwKE}}$, $\mathcal{S}$ starts simulating a new instance of the protocol for client P_i, server P_j, session identifier ssid, and CRS set as above. We will denote this instance by (P_i, ssid) and call it a *client instance.*

To simulate this instance, $\mathcal{S}$ chooses r_1, r'_1, r''_1, s_1 at random, and sets $R_1 = \mathbf{g}^{r_1}$, $S_1 = \mathbf{a}_1^{r_1}\mathbf{b}_{\mathsf{C}}^{\mu} \cdot \mathbf{g}^{r'_1}$ and $T_1 = \mathbf{g}^{r''_1}$ (note the use of arbitray constant μ instead of phash). Next, $\mathcal{S}$ generates $(\textsc{hk}_1, \textsc{hp}_1) \leftarrow \mathsf{ver.hkgen}(\textsc{crs}_{\mathsf{C}})$ and sets $i_1 = \mathcal{H}(sid, \mathsf{ssid}, P_i, P_j, R_1, S_1, \textsc{hp}_1)$. It retains $(i_1, \textsc{hk}_1)$. It then hands $(R_1, S_1, T_1, \textsc{hp}_1)$ to $\mathcal{Z}$ on behalf of this instance.

New Server Session: Sending a message to $\mathcal{Z}$. On message ($\mathsf{SvrSession}$, *sid*, ssid, P_j, P_i) from $\mathcal{F}_{\mathsf{apwKE}}$, $\mathcal{S}$ starts simulating a new instance of the protocol for client P_i, server P_j, session identifier ssid, and CRS set as above. We will denote this instance by (P_j, ssid) and call it a *server instance.*

To simulate this instance, $\mathcal{S}$ chooses r_2, r'_2, r''_2, s_2 at random, and sets $R_2 = \mathbf{g}^{r_2}$, $S_2 = \mathbf{a}_2^{r_2}\mathbf{b}_{\mathsf{S}}^{\mu} \cdot \mathbf{g}^{r'_2}$ and $T_1 = \mathbf{g}^{r''_2}$ (note the use of arbitrary constant μ instead of phash). Next, $\mathcal{S}$ generates $(\textsc{hk}_2, \textsc{hp}_2) \leftarrow \mathsf{ver.hkgen}(\textsc{crs}_{\mathsf{S}})$ and sets $i_2 = \mathcal{H}(sid, \mathsf{ssid}, P_j, P_i, R_2, S_2, \textsc{hp}_2)$. It retains $(i_2, \textsc{hk}_2)$. It then hands $(R_2, S_2, T_2, \textsc{hp}_2)$ to $\mathcal{Z}$ on behalf of this instance.

On Receiving a Message from $\mathcal{Z}$. On receiving a message $R'_2, S'_2, T'_2, \hat{\rho}'_2$ from $\mathcal{Z}$ intended for a **client instance** (P, ssid), the simulator S does the following:

1. If any of the the real world protocol checks, namely group membership and non-triviality fail it goes to the step "Other Cases" below.
2. If the message received from $\mathcal{Z}$ is same as message sent by $\mathcal{S}$ on behalf of peer P' in session ssid, then $\mathcal{S}$ just issues a NewKey call for P.

3. ("Client Only Step"): If $\mathsf{StealPwdFile}$ has already taken place then do the following: If $S_2' = R_2'^{a_2}\mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}$, then $\mathcal{S}$ calls $\mathcal{F}_{\mathsf{apwKE}}$ with $(\mathsf{Impersonate}, P, sid, \mathsf{ssid})$ and skips to the "Key Setting" step below, and otherwise go to the step "Other Cases".
4. It searches its random oracle query response pairs $\{(m_k, h_k)\}_k$ and checks whether for some $k = x$ we have $S_2' = R_2'^{a_2}\mathbf{b}_{\mathsf{S}}^{h_x}$ and m_x is of the form $(sid, P_i, P_j, \mathrm{pwd}')$. If so, then $\mathcal{S}$ calls $\mathcal{F}_{\mathsf{apwKE}}$ with $(\mathsf{TestPwd}, \mathsf{ssid}, P, \mathrm{pwd}')$ else it goes to the step "Other Cases" below. If the test passes, it sets phash $=$ h_x and goes to the "Key Setting" step below, else it goes to the step "Other Cases" below.
5. ("Key Setting Step"): Compute $i_2' = \mathcal{H}(sid, \mathsf{ssid}, P_j, P_i, R_2', S_2', \hat{\rho}_2')$.
   If $T_2' \neq \mathsf{sphf}_{\mathsf{S}}.\mathsf{privH}(\mathsf{hk}, \langle R_2', S_2'/\mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}\rangle, i_2')$ then goto the step "Other Cases". Else, compute $W_1 = \mathsf{sim}(\textsc{crs}_{\mathsf{C}}, \tau_{\mathsf{C}}, \langle R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}, T_1, i_1\rangle)$. Issue a NewKey call to $\widehat{\mathcal{F}}_{\textsc{pake}}$ with key
   $$\mathsf{ver}_{\mathsf{S}}.\mathsf{privH}(\textsc{hk}_1, \langle R_2', S_2'/\mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}, T_2', i_2'\rangle) \cdot \mathsf{ver}_{\mathsf{C}}.\mathsf{pubH}(\textsc{hp}_2', W_1)$$
6. ("Other Cases"): $\mathcal{S}$ issues a $\mathsf{TestPwd}$ call to $\widehat{\mathcal{F}}_{\textsc{pake}}$ with the dummy password μ, followed by a NewKey call with a random session key, which leads to the functionality issuing a random and independent session key to the party P.

On receiving a message $R_1', S_1', T_1', \textsc{hp}_1'$ from $\mathcal{Z}$ intended for a **server instance** (P, ssid), the response of the simulator $\mathcal{S}$ is symmetric to the response described above for client instances, except the above step "Client Only Step" is skipped.

Stealing Password File. If there was a successful online $\mathsf{TestPwd}$ call by the simulator, before this $\mathsf{StealPwdFile}$ call, the corresponding random oracle response h_k was already assigned to the variable phash. Otherwise, the simulator runs through the set of random oracle query response set of the adversary $\{(m_k, h_k)\}_k$, which were not used for an online $\mathsf{TestPwd}$ call. For all the m_k's of the form $(sid, P_i, P_j, \mathrm{pwd}')$, it calls $(\mathsf{OfflineTestPwd}, sid, \mathrm{pwd}')$. Next, $\mathcal{S}$ calls $\mathsf{StealPwdFile}$. If $\mathsf{StealPwdFile}$ returns pwd then it must equal pwd' in some m_k. Assign to the variable phash the value h_k from the earlier recorded random oracle response to m_k. Otherwise, phash is assigned a fresh random value. The Server Persistent State $\mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}$ is computed accordingly and given to the adversary.
Client Corruption. On receiving a $\mathsf{Corrupt}$ call from $\mathcal{Z}$ for client instance P_i in session ssid, the simulator $\mathcal{S}$ calls the $\mathsf{Corrupt}$ routine of $\mathcal{F}_{\mathsf{apwKE}}$ to obtain pwd. If $\mathcal{S}$ had already output a message to $\mathcal{Z}$, and not output sk_1 it computes

$$W_1 = \mathsf{sim}_{\mathsf{C}}(\textsc{crs}_{\mathsf{C}}, \tau_{\mathsf{C}}, \langle R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\mathrm{phash}}, T_1, i_1\rangle).$$

and outputs this W_1 along with pwd, and $\textsc{hk}_1$ as internal state of P_i. Note that this computation of W_1 is identical to the computation of W_1 in the computation of sk_1 (which is really output to $\mathcal{Z}$ only when $\mathrm{pwd}' = \mathrm{pwd}$).

Without loss of generality, we can assume that in the real-world if the Adversary (or Environment $\mathcal{Z}$) corrupts an instance before the session key is output then the instance does not output any session key. This is so because the Adversary (or $\mathcal{Z}$) either sets the key for that session or can compute it from the internal state it broke into.

Server Corruption. On receiving a $\mathsf{Corrupt}$ call from $\mathcal{Z}$ for server instance P_j in session ssid, the simulator $\mathcal{S}$ first performs the steps in the section on Stealing Password File above. In particular this sets the value of phash. It then calls the $\mathsf{Corrupt}$ routine of $\mathcal{F}_{\mathsf{apwKE}}$. If $\mathcal{S}$ had already output a message to $\mathcal{Z}$, and not output sk_1 it computes

$$W_2 = \mathsf{sim}_{\mathsf{S}}(\text{CRS}_{\mathsf{S}}, \tau_{\mathsf{S}}, \langle R_2, S_2/\mathbf{b}_{\mathsf{S}}^{\text{phash}}, T_2, i_2\rangle).$$

and outputs this W_2 along with HK_2 as internal state of P_j. Note that pwd is not given out.

Complexity of the simulator. Observe that on stealing the password file, the function $\mathsf{OfflineTestPwd}$ is only called once for each random oracle input, which was not already tested by calling $\mathsf{TestPwd}$. Hence the number of unique password arguments passed to $\mathsf{TestPwd}$ and $\mathsf{OfflineTestPwd}$ of $\mathcal{F}_{\mathsf{apwKE}}$ combined in the ideal world is at most the number of random oracle calls in the hybrid model.

Time complexity-wise, most of the simulator steps are $\log q$-time, where q is the security parameter. Due to Step 4 of the simulator code, where for each of the m sessions, in the worst case, it might go through all the n random oracle calls, there is an additive component of $m * n * \log q$ time. So the simulator runs in $O(mn \log q)$-time.

6.7 Proof of Indistinguishability - Series of Experiments.

We now describe a series of experiments between a probabilistic polynomial time challenger $\mathcal{C}$ and the environment $\mathcal{Z}$, starting with Expt_0 which we describe next. We will show that the view of $\mathcal{Z}$ in Expt_0 is same as its view in UC-APAKE ideal-world setting with $\mathcal{Z}$ interacting with $\mathcal{F}_{\mathsf{apwKE}}$ and the UC-PAKE simulator $\mathcal{S}$ described above in Section C.2. We end with an experiment which is identical to the real world execution of the protocol in Fig 2. We will show that the environment has negligible advantage in distinguishing between these series of experiments, leading to a proof of realization of $\mathcal{F}_{\mathsf{apwKE}}$ by the protocol $\mathit{\Pi}$.

Here is the complete code in Expt_0 (stated as it's overall experiment with $\mathcal{Z}$):

1. Responding to a random oracle query on input m: If there is a record of the form (m, r), then just return r. Otherwise, generate $r \leftarrow \mathbb{Z}_q$, record (m, r) and return r.
2. The challenger $\mathcal{C}$ picks the CRS just as in the real world, except the QA-NIZK CRS-es are generated using the crs-simulators, which also generate simulator trapdoors $\tau_{\mathsf{C}}, \tau_{\mathsf{S}}$. It retains $a_1, a_2, \tau_{\mathsf{C}}, \tau_{\mathsf{S}}, \mathsf{hk}_{\mathsf{C}}, \mathsf{hk}_{\mathsf{S}}$ as trapdoors.
   Next, (on $\mathsf{StorePwdFile}$) the challenger calls the random oracle with query $(\mathit{sid}, P_i, P_j, \text{pwd})$. It sets phash equal to the random oracle response and sets the server persistent state as $\mathbf{b}_{\mathsf{S}}^{\text{phash}}$.
   Define PhashIsSet to be true after either $\mathsf{StealPwdFile}$ has been called or the random oracle has been called with $(\mathit{sid}, P_i, P_j, \text{pwd})$ by the adversary, and false before.
   Define PwdCalled to be true after the random oracle has been called with $(\mathit{sid}, P_i, P_j, \text{pwd})$ by the adversary, and false before.

3. On receiving ($\mathsf{CltSession}$, *sid*, ssid, P_i, P_j) from $\mathcal{Z}$, $\mathcal{C}$ generates $(\textsc{hk}_1, \textsc{hp}_1) \leftarrow \mathsf{ver}_\mathsf{S}.\mathsf{hkgen}(\textsc{crs}_\mathsf{S})$. Next, $\mathcal{C}$ chooses r_1, r_1', r_1'', s_1 at random, and sets $R_1 = \mathbf{g}^{r_1}$, $S_1 = \mathbf{a}_1^{r_1}\mathbf{b}_\mathsf{C}^{\mu} \cdot \mathbf{g}^{r_1'}$ and $T_1 = \mathbf{g}^{r_1''}$. It then hands $(R_1, S_1, T_1, \textsc{hp}_1)$ to $\mathcal{Z}$ on behalf of this instance.
4. On receiving $(R_2', S_2', T_2', \textsc{hp}_2')$ from $\mathcal{Z}$, intended for client session (P_i, ssid) (and assuming no corruption of this instance):
   (a) If the received elements are either not in their respective groups, or are trivially 1, output $\mathrm{sk}_1 \leftarrow \mathbb{G}_T$.
   (b) If the message received is identical to message sent by $\mathcal{C}$ in the same session (i.e. same ssid) on behalf of the peer, then output $\mathrm{sk}_1 \leftarrow \mathbb{G}_T$ (unless the simulation of peer also received a legitimate message and its key has already been set, in which case the same key is used to output sk_1 here).
   (c) If $\textsc{PhashIsSet}$ is false, then output $\mathrm{sk}_1 \leftarrow \mathbb{G}_T$.
   (d) If $S_2' \neq R_2'^{a_2}\mathbf{b}_\mathsf{S}^{\mathrm{phash}}$, then output $\mathrm{sk}_1 \leftarrow \mathbb{G}_T$.
   (e) At this point we must have $S_2' = R_2'^{a_2}\mathbf{b}_\mathsf{S}^{\mathrm{phash}}$.
   Compute: $i_2' = \mathcal{H}(sid, \mathsf{ssid}, P_j, P_i, R_2', S_2', \textsc{hp}_2')$.
   If $T_2' \neq \mathsf{sphf}_\mathsf{S}.\mathsf{privH}(\mathsf{hk}, \langle R_2', S_2'/\mathbf{b}_\mathsf{S}^{\mathrm{phash}}\rangle, i_2')$ then output $\mathrm{sk}_1 \leftarrow \mathbb{G}_T$.
   Else, compute $W_1 = \mathsf{sim}_\mathsf{C}(\textsc{crs}_\mathsf{C}, \tau_\mathsf{C}, \langle R_1, S_1, \mathbf{b}_\mathsf{S}^{\mathrm{phash}}, T_1, i_1\rangle)$. Output:
   $$\mathrm{sk}_1 = \mathsf{ver}_\mathsf{S}.\mathsf{privH}(\textsc{hk}_1, \langle R_2', S_2'/\mathbf{b}_\mathsf{S}^{\mathrm{phash}}, T_2', i_2'\rangle) \cdot \mathsf{ver}_\mathsf{C}.\mathsf{pubH}(\textsc{hp}_2', W_1)$$
5. On a $\mathsf{Corrupt}$ call for client P_i, output pwd. If Step 3 has already happened then also output $\textsc{hk}_1$ and $W_1 = \mathsf{sim}_\mathsf{C}(\textsc{crs}_\mathsf{C}, \tau_\mathsf{C}, \langle R_1, S_1, \mathbf{b}_\mathsf{S}^{\mathrm{phash}}, T_1, i_1\rangle)$.
6. On receiving ($\mathsf{SrvSession}$, *sid*, ssid, P_j, P_i) from $\mathcal{Z}$, follow steps symmetric to Step 4, swapping subscripts and languages accordingly and replacing the condition $\textsc{PhashIsSet}$ by $\textsc{PwdCalled}$ in Step 4c.
7. On a $\mathsf{Corrupt}$ call for server P_j, if Step 3 has already happened then output $\textsc{hk}_2$, and $W_2 = \mathsf{sim}_\mathsf{S}(\textsc{crs}_\mathsf{S}, \tau_\mathsf{S}, \langle R_2, S_2/\mathbf{b}_\mathsf{S}^{\mathrm{phash}}, T_2, i_2\rangle)$. Finally, execute a $\mathsf{StealPwdFile}$ call, as described below.
8. On a $\mathsf{StealPwdFile}$ call, return $\mathbf{b}_\mathsf{S}^{\mathrm{phash}}$ as the Server Persistent State to the adversary.

All outputs of sk_1 are also accompanied with *sid*, ssid (but are not mentioned above for ease of exposition).

Note that each instance has two asynchronous phases: a phase in which $\mathcal{C}$ outputs $R_1, S_1, ...$ to $\mathcal{Z}$, and a phase where it receives a message from $\mathcal{Z}$. However, $\mathcal{C}$ cannot output sk_1 until it has completed both phases. These orderings are dictated by $\mathcal{Z}$. We will consider two different kinds of temporal orderings. A temporal ordering of different instances based on the order in which $\mathcal{C}$ outputs sk_1 in an instance will be called **temporal ordering by key output**. A temporal ordering of different instances based on the order in which $\mathcal{C}$ outputs its first message (i.e. $R_1, S_1, ...$) will be called **temporal ordering by message output**. It is easy to see that $\mathcal{C}$ can dynamically compute both these orderings by maintaining a counter (for each ordering).

We now claim that the view of $\mathcal{Z}$ in Expt_0 is statistically indistinguishable from its view in its combined interaction with $\mathcal{F}_{\mathsf{apwKE}}$ and $\mathcal{S}$. The CRS is set identically by both $\mathcal{C}$ and $\mathcal{S}$. While $\mathcal{C}$ has access to pwd from the outset and sets up the random oracle output phash corresponding to $(sid, P_i, P_j, \mathsf{ssid})$ at the beginning, $\mathcal{S}$ doesn't have access to pwd at the beginning and hence defers this step till the point where either (1) a correct online guess has been made, (2) the password file was stolen and a correct offline guess was made, (3) the client was corrupted. In all these three cases the simulator gets to know pwd and has the chance to set phash. At the point when password file is stolen, the correct pwd may not have been guessed, but phash has to be set in order to output the server persistent state. In that case $\mathcal{S}$ generates a random phash, remembers it and assigns it to the correct input when the actual password is queried. At all points, although their algorithms differ, we can see that $\mathcal{C}$ and $\mathcal{S}$ respond to random oracle queries identically.

Both $\mathcal{C}$ and $\mathcal{S}$ generate the client and server flows identically. In particular, observe that the condition PhashIsSet exactly captures the state of $\mathcal{S}$ for a client session where it knows phash and can compute the relevant elements and keys. $\mathcal{C}$ uses the condition PhashIsSet to do the same computations. Similarly for the server sessions with the condition PwdCalled. The stronger condition for the server reflects the absence of the "Client Only Step" in the server sessions simulation. In the steps where a party receives a message from the adversary, both $\mathcal{C}$ and $\mathcal{S}$ end up computing keys identically. While $\mathcal{C}$ directly checks by exponentiation with phash in the case that pwd was guessed correctly, $\mathcal{S}$ goes through the list of random oracle calls to see which response was used for exponentiation as it may not know pwd or phash at this point.

Expt_1 : In this experiment, Step 4c is removed from both client and server instances.

For client instances, observe that if the condition PwdCalled does not hold, then phash remains information theoretically unknown to the adversary. Hence the simulator code has statistically negligible chance to reach Step 4e.

For server instances (see step 6), it remains to be proven that even if the adversary steals $\mathbf{b}_{\mathsf{S}}^{\text{phash}}$, there is negligible chance of it passing the condition $S'_1 = R'^{a_1}_1 \mathbf{b}_{\mathsf{C}}^{\text{phash}}$, unless it queries the random oracle with the correct password.

This can be proved by employing DDH on $(\mathbf{b}_{\mathsf{S}}, \mathbf{b}_{\mathsf{C}}, \mathbf{b}_{\mathsf{S}}^{\text{phash}}, \mathbf{b}_{\mathsf{C}}^{\text{phash}})$. Observe that if the random oracle is not called on the correct password, then the whole experiment can be simulated without phash in the clear and just using $(\mathbf{b}_{\mathsf{S}}, \mathbf{b}_{\mathsf{C}}, \mathbf{b}_{\mathsf{S}}^{\text{phash}}, \mathbf{b}_{\mathsf{C}}^{\text{phash}})$. In particular the condition $S'_1 \overset{?}{=} R'^{a_1}_1 \mathbf{b}_{\mathsf{C}}^{\text{phash}}$ can be switched by DDH to $S'_1 \overset{?}{=} R'^{a_1}_1 \mathbf{b}_{\mathsf{C}}^{z}$, where z is independently random from phash. At this point, we see again that the adversary has statistically negligible chance of making it to Step 4e.

Once the Step 4c is removed, we switch back to the real DDH tuple, thus reaching **Expt_1**.

Expt_2 : In this experiment Step 4d is dropped altogether and Step 4e altered as follows: The condition $T'_2 \neq \mathsf{sphf_S.privH}(\mathsf{hk_S}, \langle R'_2, S'_2/\mathbf{b}_{\mathsf{S}}^{\text{phash}}\rangle, i'_2)$ in Step 4e in

Expt_0 is replaced by:

$$(S'_2 \neq R'^{a_2}_2 \mathbf{b}_{\mathsf{S}}^{\text{phash}}) \textbf{ or } (T'_2 \neq \mathsf{sphf}_{\mathsf{S}}.\mathsf{privH}(\mathsf{hk}_{\mathsf{S}}, \langle R'_2, S'_2/\mathbf{b}_{\mathsf{S}}^{\text{phash}}\rangle, i'_2)).$$

Rest of the computation of sk_1 in Step 4e remains the same.

This is just combining Steps 4d and 4e.

Expt_3 : In this experiment, in Step 4e, the condition is replaced by just $T'_2 \neq \mathsf{sphf}_{\mathsf{S}}.\mathsf{privH}(\mathsf{hk}_{\mathsf{S}}, \langle R'_2, S'_2/\mathbf{b}_{\mathsf{S}}^{\text{phash}}\rangle, i'_2)$, i.e. the disjunct $(S'_2 \neq R'^{a_2}_2 \mathbf{b}_{\mathsf{S}}^{\text{phash}})$ is dropped.

First note that T_1 is being computed randomly. The experiment Expt_3 is then statistically indistinguishable from Expt_2 by smoothness of $\mathsf{sphf}_{\mathsf{S}}$ (note that it can be shown that the polynomial number of extra bits of information leaked by the conditions $T'_2 \neq \mathsf{sphf}_{\mathsf{S}}.\mathsf{privH}(\mathsf{hk}_{\mathsf{S}}, \langle R'_2, S'_2/\mathbf{b}_{\mathsf{S}}^{\text{phash}}\rangle, i'_2)$ themselves have a negligible effect on the smoothness of $\mathsf{sphf}_{\mathsf{S}}$ – this argument is employed in the Cramer-Shoup CCA2-encryption scheme [CS02]).

Correcting Message Outputs to use pwd

Expt_4 : In this experiment the challenger in Step 3 computes S_1 in each client instance as $\mathbf{a}_1^{r_1}\mathbf{b}_{\mathsf{C}}^{\text{phash}} \cdot \mathbf{g}_1^{r'_1}$. Symmetrically, for the server instance. Note the use of phash instead of μ.

This is statistically the same, as in each instance the challenger picks a fresh and random r'_1, and it is not used anywhere else.

Expt_5 : In each instance, S_1 is computed as follows: $\mathbf{a}_1^{r_1}\mathbf{b}_{\mathsf{C}}^{\text{phash}}$. Further, T_1 is computed as follows: $T_1 = \mathsf{sphf}_{\mathsf{C}}.\mathsf{pubH}(\mathsf{hp}_{\mathsf{C}}, \langle R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\text{phash}}\rangle, i_1; r_1, \text{phash})$. Symmetrically, for the server instances.

To show that Expt_4 is computationally indistinguishable from Expt_5, we define several hybrid experiments $\mathsf{Expt}_{4,i}$ inductively. Experiment $\mathsf{Expt}_{4,0}$ is identical to Expt_4. If there are a total of N instances, $\mathsf{Expt}_{4,N}$ will be identical to Expt_5. Experiment $\mathsf{Expt}_{4,i+1}$ differs from experiment $\mathsf{Expt}_{4,i}$ in only (temporally ordered by message output) the $(i+1)$-th instance. While in $\mathsf{Expt}_{4,i}$, the $(i+1)$-th instance is simulated by $\mathcal{C}$ as in Expt_4, in $\mathsf{Expt}_{4,i+1}$ this instance is simulated as in Expt_5.

Lemma 1. *For all $i : 0 \leq i \leq N$, the view of $\mathcal{Z}$ in experiment $\mathsf{Expt}_{4,i+1}$ is computationally indistinguishable from the view of $\mathcal{Z}$ in $\mathsf{Expt}_{4,i}$.*

Proof. We define several hybrid experiments. Experiment $\mathbf{G}_0$ is identical to $\mathsf{Expt}_{4,i}$. We describe the client sessions here - the server sessions are symmetrical.

In $\mathbf{G}_1$, in the $(i+1)$-th instance T_1 is computed differently:

$$T_1 = \mathsf{sphf}_{\mathsf{C}}.\mathsf{privH}(\mathsf{hk}_{\mathsf{C}}, \langle R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\text{phash}}\rangle, i_1) \tag{1}$$

This is statistically the same as all other T_1 are either randomly computed (in instances greater than $(i+1)$), or are computed using the public hash with hp

(in instances less than $(i+1)$). Then the claim follows by smoothness of $\mathsf{sphf}_{\mathsf{C}}$, and noting that $S_1 \neq R_1^{a_1}\mathbf{b}_{\mathsf{C}}^{\text{phash}}$ in instance $(i+1)$ (by construction of $\mathsf{Expt}_{4,i}$).

In the next experiment $\mathbf{G}_2$, the challenger generates the S_1 in the $(i+1)$-th instance as follows: $S_1 = \mathbf{a}_1^{r_1}\mathbf{b}_{\mathsf{C}}^{\text{phash}}$. That the view of $\mathcal{Z}$ in experiments $\mathbf{G}_1$ and $\mathbf{G}_2$ are computationally indistinguishable follows from the DDH assumption in group $\mathbb{G}_1$ (note a_1 is not being used by the challenger, now that Step 4d is no more).

In the next experiment $\mathbf{G}_3$, change the computation of T_1 in session $(i+1)$ to use the public hash (of $\mathsf{sphf}_{\mathsf{C}}$) and witness r_1. Since, now $(R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\text{phash}})$ is in the language L_{C}, indistinguishabilty from the previous experiment follows by correctness of $\mathsf{sphf}_{\mathsf{C}}$. □

Expt_6 : In this experiment, the CRS is generated using crsgen instead of the crs-simulator, and W_1 is computed everywhere by prover of the QA-NIZK instead of the proof simulator.

Indistinguishability from the previous experiment follows by zero-knowledge property of the QA-NIZK, noting that all proofs being generated are on language memebers.

At this point, the complete experiment Expt_6 can be described as follows:

1. Responding to a random oracle query on input m: If there is a record of the form (m, r), then just return r. Otherwise, generate $r \leftarrow \mathbb{Z}_q$, record (m, r) and return r.
2. The challenger $\mathcal{C}$ picks the CRS just as in the real world. It retains $a_1, a_2, \mathsf{hk}_{\mathsf{C}}$, hk_{S} as trapdoors. Next the challenger calls the random oracle with query $(sid, P_i, P_j, \text{pwd})$. It sets phash equal to the random oracle response and sets the server persistent state as $\mathbf{b}_{\mathsf{S}}^{\text{phash}}$.
3. On receiving ($\mathsf{CltSession}$, sid, ssid, P_i, P_j) from $\mathcal{Z}$, $\mathcal{C}$ generates $(\text{HK}_1, \text{HP}_1) \leftarrow \mathsf{ver}_{\mathsf{S}}.\mathsf{hkgen}(\text{CRS}_{\mathsf{S}})$. Next, $\mathcal{C}$ chooses r_1 at random, and sets $R_1 = \mathbf{g}^{r_1}$, $S_1 = \mathbf{a}_1^{r_1}\mathbf{b}_{\mathsf{C}}^{\text{phash}}$ and $T_1 = \mathsf{sphf}_{\mathsf{C}}.\mathsf{pubH}(\mathsf{hp}_{\mathsf{C}}, \langle R_1, S_1, \mathbf{b}_{\mathsf{C}}^{\text{phash}}\rangle, i_1; r_1, \text{phash})$. It then hands $(R_1, S_1, T_1, \text{HP}_1)$ to $\mathcal{Z}$ on behalf of this instance.
4. On receiving $(R'_2, S'_2, T'_2, \text{HP}'_2)$ from $\mathcal{Z}$, intended for client session (P_i, ssid) (and assuming no corruption of this instance):

   (a) If the received elements are either not in their respective groups, or are trivially 1, output $\text{sk}_1 \leftarrow \mathbb{G}_T$.

   (b) If the message received is identical to message sent by $\mathcal{C}$ in the same session (i.e. same ssid) on behalf of the peer, then output $\text{sk}_1 \leftarrow \mathbb{G}_T$ (unless the simulation of peer also received a legitimate message and its key has already been set, in which case the same key is used to output sk_1 here).

   (e) Compute: $i'_2 = \mathcal{H}(sid, \mathsf{ssid}, P_j, P_i, R'_2, S'_2, \text{HP}'_2)$.
   If $T'_2 \neq \mathsf{sphf}_{\mathsf{S}}.\mathsf{privH}(\mathsf{hk}_{\mathsf{S}}, \langle R'_2, S'_2/\mathbf{b}_{\mathsf{S}}^{\text{phash}}\rangle, i'_2)$ then output $\text{sk}_1 \leftarrow \mathbb{G}_T$.
   Else, compute $W_1 = \mathsf{prover}_{\mathsf{C}}(\text{CRS}_{\mathsf{C}}, \langle R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\text{phash}}, T_1, i_1\rangle)$. Output:

$$\text{sk}_1 = \mathsf{ver}_{\mathsf{S}}.\mathsf{privH}(\text{HK}_1, \langle R'_2, S'_2/\mathbf{b}_{\mathsf{S}}^{\text{phash}}, T'_2, i'_2\rangle) \cdot \mathsf{ver}_{\mathsf{C}}.\mathsf{pubH}(\text{HP}'_2, W_1)$$

5. On a $\mathsf{Corrupt}$ call for client P_i, output pwd. If Step 3 has already happened then also output $\textsc{hk}_1$ and $W_1 = \mathsf{prover}_{\mathsf{C}}(\textsc{crs}_{\mathsf{C}}, \langle R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\text{phash}}, T_1, i_1\rangle)$.
6. On receiving ($\mathsf{SrvSession}$, sid, ssid, P_j, P_i) from $\mathcal{Z}$, follow steps symmetric to Step 4, swapping subscripts and languages accordingly.
7. On a $\mathsf{Corrupt}$ call for server P_j, if Step 3 has already happened then output $\textsc{hk}_2$, and $W_2 = \mathsf{prover}_{\mathsf{S}}(\textsc{crs}_{\mathsf{S}}, \langle R_2, S_2/\mathbf{b}_{\mathsf{S}}^{\text{phash}}, T_2, i_2\rangle)$. Finally, execute a $\mathsf{StealPwdFile}$ call, as described below.
8. On a $\mathsf{StealPwdFile}$ call, return $\mathbf{b}_{\mathsf{S}}^{\text{phash}}$ as the Server Persistent State to the adversary.

Handling Legitimate Messages

Expt_7 : In this experiment the Step 4b is modified as follows:
Step 4b: If the message received is identical to message sent by $\mathcal{C}$ in the same session (i.e. same ssid) on behalf of the peer, **and** if simulation of peer also received a legitimate message and its key has already been set, then output that same key here. Else, go to Step 4e.

To show that Expt_7 is indistinguishable from Expt_6 we need to go through several hybrid experiments. In each subsequent hybrid experiment one more instance is modified, and the order in which these instances are handled is determined by temporal order of key output. In the hybrid experiment $\mathsf{Expt}_{6,i}$ $(N \geq i \geq 1)$, the Step 3(b) in the i-th temporally ordered instance is modified as required in Expt_7 description above. Experiment $\mathsf{Expt}_{6,0}$ is same as experiment Expt_6, and experiment $\mathsf{Expt}_{6,N}$ is same as experiment Expt_7.

Lemma 2. *For all $i \in [1..N]$, experiment $\mathsf{Expt}_{6,i}$ is computationally indistinguishable from $\mathsf{Expt}_{6,i-1}$.*

Proof. The lemma is proved using several hybrid experiments of its own. The experiment $\mathbf{H}_0$ is same as $\mathsf{Expt}_{6,i-1}$.

In experiment $\mathbf{H}_1$ the CRS is set as in the real world, except that the QA-NIZK $\textsc{crs}_{\mathsf{C}}$ is set using the crs simulators $\mathsf{crssim}_{\mathsf{C}}$ (the challenger retains the trapdoors τ_{C} output by the crs simulator). All proofs W_1 are still computed using $\mathsf{prover}_{\mathsf{C}}$. Experiments $\mathbf{H}_0$ and $\mathbf{H}_1$ are indistinguishable as the QA-NIZK has the property that the simulated CRS and the real-world CRS are statistically identical.

In experiment $\mathbf{H}_2$, in instance i, the value W_1 (in Step 4e or corruption) is generated using the proof simulator using trapdoor τ. Indistinguishability follows by zero-knowledge property of the QA-NIZK as the proof being generated is on a language member.

In experiment $\mathbf{H}_3$, in instance i, the value T_1 is generated using the private hash key hk_{C}, and the private hash function $\mathsf{sphf}_{\mathsf{C}}.\mathsf{privH}$ (thus eliminating the use of witness r_1). Experiments $\mathbf{H}_2$ and $\mathbf{H}_3$ are indistinguishable by the correctness of $\mathsf{sphf}_{\mathsf{C}}$.

In experiment $\mathbf{H}_4$, in instance i, the values R_1, S_1 are generated as $R_1 = \mathbf{g}^{r_1}$, $S_1 = \mathbf{a}_1^{r_1}\mathbf{b}_{\mathsf{C}}^{\text{phash}} \cdot \mathbf{g}^{r_1'}$. where r_1, r_1' are random and independent. This follows by employing DDH on $\mathbf{g}, \mathbf{g}^{r_1}, \mathbf{a}_1$ and either $\mathbf{g}^{a_1 r_1}$ or $\mathbf{g}^{a_1 r_1 + r_1'}$.

In experiment $\mathbf{H}_5$, in peer of instance i, in Step 4e the condition $T'_1 \neq \mathsf{sphf}_{\mathsf{C}}.\mathsf{privH}(\mathsf{hk}_{\mathsf{C}}, \langle R'_1, S'_1, \mathbf{b}_{\mathsf{S}}^{\text{phash}}\rangle, i'_1)$ is replaced by $(S'_1 \neq R_1'^{a_1}\mathbf{b}_{\mathsf{C}}^{\text{phash}})$ **or** $T'_1 \neq \mathsf{sphf}_{\mathsf{C}}.\mathsf{privH}(\mathsf{hk}_{\mathsf{C}}, \langle R'_1, S'_1, \mathbf{b}_{\mathsf{S}}^{\text{phash}}\rangle, i'_1)$. Indistinguishability from experiment $\mathbf{H}_4$ follows by smooth$_2$ property of $\mathsf{sphf}_{\mathsf{C}}$, noting that at most one bad $\mathsf{sphf}_{\mathsf{C}}.\mathsf{privH}$ is being output to the Adversary (namely T_1 in instance i).

In experiment $\mathbf{H}_6$, in instance i, change Step 4b as follows: If the message received is identical to message sent by $\mathcal{C}$ in the same instance (i.e. same SSID) on behalf of the peer,

- If simulation of peer also received a legitimate message and its key has already been set, then output that same key here. If peer is corrupted, output the key supplied by the Adversary.
- Else, compute $i'_2 = \mathcal{H}(sid, \mathsf{ssid},\ P_j, P_i,\ R'_2, S'_2, \textsc{hp}'_2)$, Output

$$\mathsf{ver}_{\mathsf{S}}.\mathsf{privH}(\textsc{hk}_1, \langle R'_2, S'_2/\mathbf{b}_{\mathsf{S}}^{\text{phash}}, T'_2\rangle, i'_2) \cdot \mathsf{ver}_{\mathsf{C}}.\mathsf{privH}(\textsc{hk}_2, \langle R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\text{phash}}, T_1\rangle, i_1)$$

Here $\textsc{hk}_2$ is the $\textsc{hk}$ output by $\mathsf{ver}_{\mathsf{S}}.\mathsf{hkgen}$ in the peer instance of instance i.

The experiments $\mathbf{H}_6$ and $\mathbf{H}_5$ are computationally indistinguishable by noting the following three facts:

1. In the peer of instance of instance i (which generated $\textsc{hk}_2$), in Step 4e the computation $\mathsf{ver}_{\mathsf{C}}.\mathsf{privH}(\textsc{hk}_2, \cdot)$ is on a language member, as this computation is only reached if the the incoming tuple is in the language.
2. Also, note that only one QA-NIZK proof is being simulated and that is in this same instance, but in a mutually exclusive step (Step 4e or corruption). Moreover, the CRS generated by the crs simulator is statistically identical to the CRS generated by $\mathsf{crsgen}_{\mathsf{C}}$.
3. Then, $\mathsf{ver}_{\mathsf{C}}.\mathsf{privH}(\textsc{hk}_2, \langle R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\text{phash}}, T_1\rangle, i_1)$ is random even when the adversary is given $\textsc{hp}_2$ by smoothness of the QA-NIZK, since $S_1 \neq R_1^{a_1}\mathbf{b}_{\mathsf{C}}^{\text{phash}}$.

In experiment $\mathbf{H}_7$, in peer of instance i, in Step 4e the condition "$(S'_1 \neq R_1'^{a_1}\mathbf{b}_{\mathsf{C}}^{\text{phash}})$ **or** $T'_1 \neq \mathsf{sphf}_{\mathsf{C}}.\mathsf{privH}(\mathsf{hk}_{\mathsf{C}}, \langle R'_1, S'_1, \mathbf{b}_{\mathsf{C}}^{\text{phash}}\rangle, i'_1)$" is replaced by "$T'_1 \neq \mathsf{sphf}_{\mathsf{C}}.\mathsf{privH}(\mathsf{hk}_{\mathsf{C}}, \langle R'_1, S'_1, \mathbf{b}_{\mathsf{C}}^{\text{phash}}\rangle, i'_1)$". Indistinguishability from experiment $\mathbf{H}_6$ follows by smooth$_2$ property of the $\mathsf{sphf}_{\mathsf{C}}$, noting that at most one bad $\mathsf{sphf}_{\mathsf{C}}.\mathsf{privH}$ is being output to the Adversary (namely T_1 in instance i).

In experiment $\mathbf{H}_8$, in instance i, R_1, S_1 are generated as $R_1 = \mathbf{g}^{r_1}$, $S_1 = \mathbf{a}_1^{r_1}\mathbf{b}_{\mathsf{C}}^{\text{phash}}$, by employing DDH.

In experiment $\mathbf{H}_9$, in instance i, T_1 is generated using the public hash key hp_{C}, and witness r_1. Indistinguishability follows by correctness of the sphf.

In experiment $\mathbf{H}_{10}$, the QA-NIZK is generated using the real world CRS generator. Moreover, in instance i, in Step 4e and corruption step, W_1 is computed using the real world prover. Indistinguishability follows by zero-knowledge property of the QA-NIZK.

In experiment $\mathbf{H}_{11}$, in Step 4b the key is output as follows:

- Else, compute $i'_2 = \mathcal{H}(sid, \mathsf{ssid},\ P_j, P_i,\ R'_2, S'_2, \textsc{hp}'_2)$.
  Compute $W_1 = \mathsf{prover}_{\mathsf{C}}(\textsc{crs}_{\mathsf{C}}, \langle R_1, S_1, \mathbf{b}_{\mathsf{S}}^{\text{phash}}, T_1, i_1\rangle, r_1)$. Output

$$\mathsf{ver}_{\mathsf{S}}.\mathsf{privH}(\textsc{hk}_1, \langle R'_2, S'_2/\mathbf{b}_{\mathsf{S}}^{\text{phash}}, T'_2\rangle, \iota'_2) \cdot \mathsf{ver}_{\mathsf{C}}.\mathsf{pubH}(\textsc{hp}'_2, W_1)$$

Indistinguishability follows by noting that $\textsc{hp}'_2$ is exactly the $\textsc{hp}_2$ computed by the challenger in the peer instance. The claim then follows by completeness of the smooth QA-NIZK.

The induction step is complete now, as the above computation of the session key is same as in Step 4e. □

Handling Adversarial Messages

$\textbf{\textsf{Expt}}_8$: In this experiment in Step 4e the condition is changed to "$(S'_2 \neq R'^{a_2}_2 \mathbf{b}^{\text{phash}}_{\mathsf{S}})$ **or** $T'_2 \neq \mathsf{sphf}_{\mathsf{S}}.\mathsf{privH}(\mathsf{hk}_{\mathsf{S}}, \langle R'_2, S'_2/\mathbf{b}^{\text{phash}}_{\mathsf{S}}\rangle, i'_2)$". In other words, the disjunct $(S'_2 \neq R'^{a_2}_2 \mathbf{b}^{\text{phash}}_{\mathsf{S}})$ is introduced.

Indistinguishability follows by the same argument as employed in experiments Expt_3 and Expt_2.

$\textbf{\textsf{Expt}}_9$: In this experiment Step 4e is dropped altogether.

We first show that if the condition:

$$(S'_2 \neq R'^{a_2}_2 \mathbf{b}^{\text{phash}}_{\mathsf{S}}) \textbf{ or } T'_2 \neq \mathsf{sphf}_{\mathsf{S}}.\mathsf{privH}(\mathsf{hk}_{\mathsf{S}}, \langle R'_2, S'_2/\mathbf{b}^{\text{phash}}_{\mathsf{S}}\rangle, i'_2)$$

holds, then $(R'_2, S'_2/\mathbf{b}^{\text{phash}}_{\mathsf{S}}, T'_2, i'_2)$ is not in language L^+_{S} (for which the QA-NIZK is defined). Clearly, if the first disjunct does not hold then the tuple is not in the language. So, suppose $(S'_2 = R'^{a_2}_2 \mathbf{b}^{\text{phash}}_{\mathsf{S}})$, with witness r_2 for R'_2. Then, by correctness of the sphf,

$$\mathsf{sphf}_{\mathsf{S}}.\mathsf{privH}(\mathsf{hk}_{\mathsf{S}}, \langle R'_2, S'_2/\mathbf{b}^{\text{phash}}_{\mathsf{S}}\rangle, i'_2) = \mathsf{sphf}_{\mathsf{S}}.\mathsf{pubH}(\mathsf{hp}_{\mathsf{S}}, \langle R'_2, S'_2/\mathbf{b}^{\text{phash}}\rangle, i'_2; r_2).$$

Therefore, again, the tuple is not in the language.

Thus, $\mathsf{ver}_{\mathsf{S}}.\mathsf{privH}(\textsc{hk}_1, \langle R'_2, S'_2/\mathbf{b}^{\text{phash}}_{\mathsf{S}}, T'_2\rangle, i'_2)$ is random, even when the Adversary is given $\textsc{hp}_1$, by smooth-soundness of the QA-NIZK.

$\textbf{\textsf{Expt}}_{10}$: In this experiment the Step 4b is dropped. In other words, the challenger code goes straight from Step 4a to Step 4e.

Experiments Expt_{10} and Expt_9 produce the same view for $\mathcal{Z}$, since if both peers (of a instance) received legitimate messages forwarded by $\mathcal{Z}$, then Step 4e computes the same instance key in both instances.

Finally, a simple examination shows that the view of $\mathcal{Z}$ in Expt_{10} is identical to the real world protocol. That completes the proof of Theorem 3. □

References

[BBC+13] Fabrice Benhamouda, Olivier Blazy, Céline Chevalier, David Pointcheval, and Damien Vergnaud. New techniques for SPHFs and efficient one-round PAKE protocols. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 449–475. August 2013.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 41–55. August 2004.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications (extended abstract). In *20th ACM STOC*, pages 103–112. ACM Press, May 1988.

[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.

[BM93] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In V. Ashby, editor, *ACM CCS 93*, pages 244–250. ACM Press, November 1993.

[BMP00] Victor Boyko, Philip D. MacKenzie, and Sarvar Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 156–171. May 2000.

[Boy09] Xavier Boyen. HPAKE: Password authentication secure against cross-site user impersonation. In Juan A. Garay, Atsuko Miyaji, and Akira Otsuka, editors, *CANS 09*, volume 5888 of *LNCS*, pages 279–298. December 2009.

[BP13] Fabrice Benhamouda and David Pointcheval. Verifier-based password-authenticated key exchange: New models and constructions. *IACR Cryptology ePrint Archive*, 2013:833, 2013.

[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. May 2000.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In V. Ashby, editor, *ACM CCS 93*, pages 62–73. ACM Press, November 1993.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[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. May 2005.

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

[CS02] Ronald Cramer and Victor Shoup. Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 45–64. April / May 2002.

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

[EHK$^+$13] Alex Escala, Gottfried Herold, Eike Kiltz, Carla Ràfols, and Jorge Villar. An algebraic framework for Diffie-Hellman assumptions. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part II*, volume 8043 of *LNCS*, pages 129–147. August 2013.

[FLR$^+$10] Marc Fischlin, Anja Lehmann, Thomas Ristenpart, Thomas Shrimpton, Martijn Stam, and Stefano Tessaro. Random oracles with(out) programmability. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 303–320. December 2010.

[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. August 2006.

[GS08] Jens Groth and Amit Sahai. Efficient non-interactive proof systems for bilinear groups. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 415–432. April 2008.

[HK98] Shai Halevi and Hugo Krawczyk. Public-key cryptography and password protocols. In *ACM CCS 98*, pages 122–131. ACM Press, November 1998.

[JG04] Shaoquan Jiang and Guang Gong. Password based key exchange with mutual authentication. In Helena Handschuh and Anwar Hasan, editors, *SAC 2004*, volume 3357 of *LNCS*, pages 267–279. August 2004.

[JR12] Charanjit S. Jutla and Arnab Roy. Relatively-sound NIZKs and password-based key-exchange. In Marc Fischlin, Johannes Buchmann, and Mark Manulis, editors, *PKC 2012*, volume 7293 of *LNCS*, pages 485–503. May 2012.

[JR13] Charanjit S. Jutla and Arnab Roy. Shorter quasi-adaptive NIZK proofs for linear subspaces. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part I*, volume 8269 of *LNCS*, pages 1–20. December 2013.

[JR14] Charanjit S. Jutla and Arnab Roy. Switching lemma for bilinear tests and constant-size NIZK proofs for linear subspaces. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 295–312. August 2014.

[JR15] Charanjit S. Jutla and Arnab Roy. Dual-system simulation-soundness with applications to UC-PAKE and more. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 630–655. November / December 2015.

[KV11] Jonathan Katz and Vinod Vaikuntanathan. Round-optimal password-based authenticated key exchange. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 293–310. March 2011.

[KW15] Eike Kiltz and Hoeteck Wee. Quasi-adaptive NIZK for linear subspaces revisited. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 101–128. April 2015.

[LPJY14] Benoît Libert, Thomas Peters, Marc Joye, and Moti Yung. Non-malleability from malleability: Simulation-sound quasi-adaptive NIZK proofs and CCA2-secure encryption from homomorphic signatures. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 514–532. Springer, Heidelberg, May 2014.

[Mac01] Philip D. MacKenzie. More efficient password-authenticated key exchange. In David Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 361–377. April 2001.

[Wat09] Brent Waters. Dual system encryption: Realizing fully secure IBE and HIBE under simple assumptions. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 619–636. August 2009.

A Proof of Theorem 1

Theorem 1. (re-stated) If a matrix distribution $\mathcal{D}_k$ on $\mathbb{Z}_q^{(k+1)\times k}$ is boostable to a matrix distribution $\mathcal{D}_{l,k}$ on $\mathbb{Z}_q^{l\times k}$ then the $\mathcal{D}_k$-MDDH assumption implies the $\mathcal{D}_{l,k}$-MDDH assumption.

Proof. We prove this by a sequence of hybrids, where in the i-th hybrid we transform row $k+i$ from that of $[\mathbf{Bs}]$ to uniformly random. We start off with $i=0$, where we have the real output $[\mathbf{Bs}]$ and end with $i=l-k$ where we have the fake output which is uniformly random in $\mathbb{Z}_q^l$.

The i-th hybrid $([\mathbf{B}],[\mathbf{b}])$ is computed as follows. We sample $[\mathbf{A}]$ from $\mathcal{D}_k$ and $\mathbf{s}$ from $\mathbb{Z}_q^k$. We set $[\bar{\mathbf{B}}]$ as $[\bar{\mathbf{A}}]$ and, if $i\neq 0$, the row i of $[\underline{\mathbf{B}}]$ as the row i of $\mathbf{F}[\mathbf{A}]$. All other rows $j\neq i$ of $[\underline{\mathbf{B}}]$ are set to the j-th row of $\mathbf{E}[\bar{\mathbf{A}}]$. We set the top k elements of $[\mathbf{b}]$ to be $[\bar{\mathbf{A}}\mathbf{s}]$ and choose all the $(k+j)$-th elements, where $j<i$, of $[\mathbf{b}]$ uniformly at random from $\mathbb{Z}_q$. If $i\neq 0$, we set the $(k+i)$-th element of $[\mathbf{b}]$ to be the i-th element of $\mathbf{F}[\mathbf{As}]$. For all $j>i$, we set the $(k+j)$-th element of $[\mathbf{b}]$ to be the j-th element of $\mathbf{E}[\bar{\mathbf{A}}\mathbf{s}]$. To summarize, $[\mathbf{b}]$ is computed as:

$$\begin{bmatrix} [\bar{\mathbf{A}}\mathbf{s}] \\ \$ \\ \vdots \\ \$ \\ (\mathbf{F}[\mathbf{As}])_i \\ (\mathbf{E}[\bar{\mathbf{A}}\mathbf{s}])_{j=(i+1)\text{ to }(l-k)} \end{bmatrix}$$

We observe that the 0-th hybrid has the distribution of $([\mathbf{B}],[\mathbf{Bs}])$ and the $(l-k)$-th hybrid has the distribution of $([\mathbf{B}],[\mathbf{s}'])$, with $\mathbf{s}'$ uniform in $\mathbb{Z}_q^l$.

Now, $(\mathbf{F}[\mathbf{As}])_i = (\mathbf{F}_l)_i[\bar{\mathbf{A}}\mathbf{s}] + (\mathbf{F}_r)_i[\underline{\mathbf{A}}\mathbf{s}]$, where $\mathbf{F}_l$ is the first k-column submatrix of $\mathbf{F}$ and $\mathbf{F}_r$ is the last column of $\mathbf{F}$. Suppose we are given a $\mathcal{D}_k$-MDDH challenge $([\mathbf{A}],\boldsymbol{\chi} = [\mathbf{As}]\ \textbf{or}\ [\mathbf{s}'])$. If $\boldsymbol{\chi} = [\mathbf{As}]$, then $(\mathbf{F}\boldsymbol{\chi})_i$ is distributed as $(\mathbf{F}[\mathbf{As}])_i$. Else, if $\boldsymbol{\chi} = [\mathbf{s}']$, then $(\mathbf{F}\boldsymbol{\chi})_i$ is distributed uniformly randomly in $\mathbb{Z}_q$, since $(\mathbf{F}_r)_i$ is overwhelmingly non-zero by design. Next we transition to an intermediate hybrid i' where $[\mathbf{b}]$ is computed as:

$$\begin{bmatrix} [\bar{\mathbf{A}}\mathbf{s}] \\ \$ \\ \vdots \\ \$ \\ \$ \\ (\mathbf{E}[\bar{\mathbf{A}}\mathbf{s}])_{j=(i+1)\text{ to }(l-k)} \end{bmatrix}$$

As shown above, the hybrid i' is indistinguishable from hybrid i by the $\mathcal{D}_k$-MDDH assumption. Next we transition to the hybrid $i+1$ where $[\mathbf{b}]$ is computed as:

$$\begin{bmatrix} [\bar{\mathbf{A}}\mathrm{s}] \\ \$ \\ \vdots \\ \$ \\ \$ \\ (\mathbf{F}[\mathbf{A}\mathrm{s}])_{(i+1)} \\ (\mathbf{E}[\bar{\mathbf{A}}\mathrm{s}])_{j=(i+2)\text{ to }(l-k)} \end{bmatrix}$$

The hybrid $i+1$ is indistinguishable from hybrid i', as $\mathbf{E}\bar{\mathbf{A}}$ is identically distributed as $\mathbf{FA}$. The theorem is thus established by chaining all the hybrids.

B Hardness Assumptions

Definition 3 (DDH [DH76]). *Assuming a generation algorithm $\mathcal{G}$ that outputs a tuple $(q, \mathbb{G}, \mathbf{g})$ such that $\mathbb{G}$ is of prime order q and has generator g, the DDH assumption asserts that it is computationally infeasible to distinguish between $(\mathbf{g}, a\cdot\mathbf{g}, b\cdot\mathbf{g}, c\cdot\mathbf{g})$ and $(\mathbf{g}, a\cdot\mathbf{g}, b\cdot\mathbf{g}, ab\cdot\mathbf{g})$ for $a, b, c \leftarrow \mathbb{Z}_q$. More formally, for all PPT adversaries A there exists a negligible function $\nu()$ such that*

$$\left| \begin{matrix} \Pr[(q, \mathbb{G}, \mathbf{g}) \leftarrow \mathcal{G}(1^m); a, b, c \leftarrow \mathbb{Z}_q : A(\mathbf{g}, a\cdot\mathbf{g}, b\cdot\mathbf{g}, c\cdot\mathbf{g}) = 1] - \\ \Pr[(q, \mathbb{G}, \mathbf{g}) \leftarrow \mathcal{G}(1^m); a, b \leftarrow \mathbb{Z}_q : A(\mathbf{g}, a\cdot\mathbf{g}, b\cdot\mathbf{g}, ab\cdot\mathbf{g}) = 1] \end{matrix} \right| < \nu(m)$$

Note that this is a $\mathcal{D}_1$-MDDH assumption with the matrix $\mathbf{A}$ being the 2×1 matrix which is the transpose of $(a\ 1)$.

Definition 4 (SXDH [BBS04]). *Consider a generation algorithm $\mathcal{G}$ taking the security parameter as input, that outputs a tuple $(q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, \mathbf{g}_1, \mathbf{g}_2)$, where $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$ are groups of prime order q with generators $\mathbf{g}_1, \mathbf{g}_2$ and $e(\mathbf{g}_1, \mathbf{g}_2)$ respectively and which allow an efficiently computable $\mathbb{Z}_q$-bilinear pairing map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. The Symmetric eXternal decisional Diffie-Hellman (SXDH) assumption asserts that the Decisional Diffie-Hellman (DDH) problem is hard in* both *the groups $\mathbb{G}_1$ and $\mathbb{G}_2$.*

C Single-Round UC Password-Based Key Exchange

The essential elements of the Universal Composability framework can be found in [Can01]. In the following, we adopt the definition for password-based key exchange (UC-PAKE) from Canetti et al [CHK$^+$05].

C.1 UC-PAKE Definition

Just as in the normal key-exchange functionality, if both participating parties are not corrupted, then they receive the same uniformly distributed session key and the adversary learns nothing of the key except that it was generated. However, if one of the parties is corrupted, then the adversary determines the session

Functionality $\mathcal{F}_{\mathbf{pake}}$

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

Upon receiving a query $(\mathsf{NewSession}, sid, P_i, P_j, pw, role)$ **from party** P_i**:** Send $(\mathsf{NewSession}, sid, P_i, P_j, role)$ to S. In addition, if this is the first $\mathsf{NewSession}$ query, or if this is the second $\mathsf{NewSession}$ query and there is a record (P_j, P_i, pw'), then record (P_i, P_j, pw) and mark this record fresh.

Upon receiving a query $(\mathsf{TestPwd}, sid, P_i, pw')$ **from the adversary** S**:** If there is a record of the form (P_i, P_j, pw) which is fresh, then do: If $pw = pw'$, mark the record $\mathsf{compromised}$ and reply to S with "correct guess". If $pw \neq pw'$, mark the record $\mathsf{interrupted}$ and reply with "wrong guess".

Upon receiving a query $(\mathsf{NewKey}, sid, P_i, sk)$ **from** S**, where** $|sk| = k$**:** If there is a record of the form (P_i, P_j, pw), and this is the first NewKey query for P_i, then:

- If this record is $\mathsf{compromised}$, or either P_i or P_j is corrupted, then output (sid, sk) to player P_i.
- If this record is fresh, and there is a record (P_j, P_i, pw') with $pw' = pw$, and a key sk' was sent to P_j, and (P_j, P_i, pw) was fresh at the time, then output (sid, sk') to P_i.
- In any other case, pick a new random key sk' of length k and send (sid, sk') to P_i.

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

Upon receiving $(\mathsf{Corrupt}, sid, P_i)$ **from** S**:** if there is a (P_i, P_j, pw) recorded, return pw to S, and mark P_i corrupted.

Fig. 3. The password-based key-exchange functionality $\mathcal{F}_{\text{PAKE}}$

key. This power to the adversary is *also* given in case it succeeds in guessing the parties' shared password. Participants also detect when the adversary makes an unsuccessful attempt. If the adversary makes a wrong password guess in a given session, then the session is marked $\mathsf{interrupted}$ and the parties are provided random and independent session keys. If however the adversary makes a successful guess, then the session is marked $\mathsf{compromised}$, and the adversary is allowed to set the session key. If a session remains marked fresh, meaning that it is neither interrupted nor compromised. uncorrupted parties conclude with both parties receiving the same, uniformly distributed session key.

The formal description of the UC-PAKE functionality $\mathcal{F}_{\text{PAKE}}$ is given in Figure 3.

The real-world protocol we provide is also shown to be secure when different sessions use the same common reference string (CRS) To achieve this goal, we consider the *Universal Composability with joint state* (JUC) formalism of Canetti and Rabin [CR03]. This formalism provides a "wrapper layer" that deals with "joint state" among different copies of the protocol. In particular, defining a functionality $\mathcal{F}$ also implicitly defines the multi-session extension of $\mathcal{F}$ (denoted by $\hat{\mathcal{F}}$): $\hat{\mathcal{F}}$ runs multiple independent copies of $\mathcal{F}$, where the copies are distinguished via sub-session IDs ssid. The JUC theorem [CR03] asserts that for any protocol π that uses multiple independent copies of $\mathcal{F}$, composing π instead with a single copy of a protocol that realizes $\hat{\mathcal{F}}$, preserves the security of π.

Generate $\mathbf{g}_1 \leftarrow \mathbb{G}_1, \mathbf{g}_2 \leftarrow \mathbb{G}_2$ and $\mathbf{a} = \mathbf{g}_1^a$ with $a \leftarrow \mathbb{Z}_q$ as DH parameters ρ.
Let $\mathcal{H}$ be a CRHF, and sphf be a smooth$_2$ SPHF family for the DH family.
$(\mathsf{hp}, \mathsf{hk}) \leftarrow \mathsf{sphf.hkgen}(\rho)$.
Let $(\mathsf{pargen}, \mathsf{crsgen}, \mathsf{prover}, \mathsf{ver})$ be a Smooth QA-NIZK for language L,
$L = \{R, S, T, l : \exists r, R = \mathbf{g}_1^r, S = \mathbf{a}^r, T = \mathsf{sphf.pubH}(\mathsf{hp}, \langle R, S\rangle, l; r)\}$.
$\textsc{crs} \leftarrow \mathsf{crsgen}(\rho)$.

$$\text{CRS} := (\rho, \mathsf{hp}, \textsc{crs}, \mathcal{H}).$$

Party P_i	Network
Input $(\texttt{NewSession}, sid, \mathsf{ssid}, P_i, P_j, \text{pwd}, initiator/responder)$ Choose $r_1 \xleftarrow{\$} \mathbb{Z}_q$, $(\textsc{hk}_1, \textsc{hp}_1) \leftarrow \mathsf{ver.hkgen}(\textsc{crs})$. Set $R_1 = \mathbf{g}_1^{r_1}$, $S_1 = \text{pwd} \cdot \mathbf{a}^{r_1}$, $T_1 = \mathsf{sphf.pubH}(\mathsf{hp}, \langle R, S_1/\text{pwd}\rangle, i_1; r_1)$, $W_1 = \mathsf{prover}(\textsc{crs}, \langle R_1, S_2, T_1, i_1\rangle; r_1)$, where $i_1 = \mathcal{H}(sid, \mathsf{ssid}, P_i, P_j, R_1, S_1, \textsc{hp}_1)$. Erase r_1, send $(R_1, S_1, T_1, \textsc{hp}_1)$ and retain $(W_1, \textsc{hk}_1)$.	$\xrightarrow{R_1, S_1, T_1, \text{HP}_1} P_j$
Receive $R'_2, S'_2, T'_2, \textsc{hp}'_2$. If any of $R'_2, S'_2, T'_2, \textsc{hp}'_2$ is not in their respective group or is 1, set $\text{sk}_1 \xleftarrow{\$} \mathbb{G}_T$, else compute $i'_2 = \mathcal{H}(sid, \mathsf{ssid}, P_j, P_i, R'_2, S'_2, \textsc{hp}'_2)$, Compute $\text{sk}_1 = \mathsf{ver.privH}(\textsc{hk}_1, \langle R'_2, S'_2/\text{pwd}, T'_2, i'_2\rangle) \cdot \mathsf{ver.pubH}(\textsc{hp}'_2, W_1)$. Output $(sid, \mathsf{ssid}, \text{sk}_1)$.	$\xleftarrow{R'_2, S'_2, T'_2, \text{HP}'_2} P_j$

Fig. 4. Single-round UC-PAKE protocol under SXDH assumption.

C.2 Proof of Realization of the UC-PAKE Functionality

In this section we state and prove that the protocol in Fig. 4 realizes the multi-session ideal functionality $\widehat{\mathcal{F}}_{\text{PAKE}}$.

Theorem 4. *Assuming the existence of SXDH-hard groups, the protocol in Fig 4 securely realizes the $\widehat{\mathcal{F}}_{\text{PAKE}}$ functionality in the $\mathcal{F}_{\text{CRS}}$ hybrid model, in the presence of adaptive corruption adversaries.*

We start by defining the UC simulator in detail.

The Simulator for the UC Protocol. We will assume that the adversary $\mathcal{A}$ in the UC protocol is dummy, and essentially passes back and forth commands and messages from the environment $\mathcal{Z}$. Thus, from now on we will use environment $\mathcal{Z}$ as the real adversary, which outputs a single bit. The simulator $\mathcal{S}$ will be the ideal world adversary for $\widehat{\mathcal{F}}_{\text{PAKE}}$. It is a universal simulator that uses $\mathcal{A}$ as a black box.

For each instance (session and a party), we will use subscript 2 along with a prime, to refer to variables received in the message from $\mathcal{Z}$ (i.e $\mathcal{A}$), and use subscript 1 to refer to variables computed in the instance under consideration. We will call a message *legitimate* if it was not altered by $\mathcal{Z}$, and delivered in the correct session and to the correct party.

The simulator $\mathcal{S}$ picks the CRS just as in the real world, except the QA-NIZK $\textsc{crs}$ is generated using the crs-simulator, which also generates a simulator trapdoor τ. It retains a, τ, hk as trapdoors.

The next main difference in the simulation of the real world parties is that $\mathcal{S}$ uses a dummy message μ instead of the real password which it does not have access to. Further, it decrypts the incoming message R_2', S_2', T_2' to compute a pwd$'$, which it uses to call the ideal functionality's test function. If the test succeeds, it produces a sk (see below) and sends it to the ideal functionality to be output to the party concerned.

New Session: Sending a message to $\mathcal{Z}$. On message ($\mathsf{NewSession}$, sid, ssid, i, j, role) from $\widehat{\mathcal{F}}_{\text{PAKE}}$, S starts simulating a new instance of the protocol $\varPi$ for party P_i, peer P_j, session identifier ssid, and CRS set as above. We will denote this instance by (P_i, ssid). To simulate this instance, S chooses r_1, r_1', r_1'' at random. Also, $(\textsc{hk}_1, \textsc{hp}_1) \leftarrow \mathsf{ver.hkgen}(\textsc{crs})$. It sets $R_1 = \mathbf{g}_1^{r_1}$, $S_1 = \mu \cdot \mathbf{a}^{r_1} \cdot \mathbf{g}_1^{r_1'}$, $T_1 = \mathbf{g}_1^{r_1''}$. Let $\iota_1 = \mathcal{H}(sid, \mathsf{ssid},\ P_i, P_j,\ R_1, S_1, \textsc{hp}_1)$. (Note the use of μ instead of pwd).

It retains $r_1, r_1', r_1'', \iota_1, \textsc{hk}_1$ (and μ if chosen randomly). It then hands R_1, S_1, T_1, $\textsc{hp}_1$ to $\mathcal{Z}$ on behalf of this instance.

On Receiving a Message from $\mathcal{Z}$. On receiving a message $R_2', S_2', T_2', \textsc{hp}_2'$ from $\mathcal{Z}$ intended for this instance (P_i, ssid), the simulator S makes the real world protocol checks, namely group membership and non-triviality. If any of these checks fail, it issues a $\mathsf{TestPwd}$ call to $\widehat{\mathcal{F}}_{\text{PAKE}}$ with the dummy password μ, followed by a NewKey call with a random session key, which leads to the functionality issuing a random and independent session key to the party P_i (regardless of whether the instance was interrupted or compromised).

Otherwise, if the message received from $\mathcal{Z}$ is same as message sent by $\mathcal{S}$ on behalf of peer P_j in session ssid, then $\mathcal{S}$ just issues a NewKey call for P_i.

Else, it computes pwd$'$ by decrypting S_2', i.e. setting it to $S_2'/(R_2')^a$. $\mathcal{S}$ then calls $\widehat{\mathcal{F}}_{\text{PAKE}}$ with ($\mathsf{TestPwd}$, ssid, P_i, pwd$'$). Regardless of the reply from $\mathcal{F}$, it then issues a NewKey call for P_i with key computed as follows (recall, $R_1, S_1, \iota_1, r_1', r_1''$ from earlier in this instance when the message was sent to $\mathcal{Z}$). Let,

$$\begin{aligned}
\iota_2' &= \mathcal{H}(sid, \mathsf{ssid}, P_j, P_i, R_2', S_2', \textsc{hp}_2'),\\
W_1 &= \mathsf{sim}(\textsc{crs}, \tau, \langle R_1, S_1/\text{pwd}', T_1, \iota_1\rangle)
\end{aligned}$$

If $T_2' \neq \mathsf{sphf.privH}(\mathsf{hk}, \langle R_2', (R_2')^a\rangle, \iota_2')$ then call $\widehat{\mathcal{F}}_{\text{PAKE}}$'s NewKey with a random key else call NewKey with key

$$\mathsf{ver.privH}(\textsc{hk}_1, \langle R_2', (R_2')^a, T_2', \iota_2'\rangle) \cdot \mathsf{ver.pubH}(\textsc{hp}_2', W_1).$$

By definition of $\widehat{\mathcal{F}}_{\text{PAKE}}$, this has the effect that if the pwd$'$ was same as the actual pwd previously recorded in $\widehat{\mathcal{F}}_{\text{PAKE}}$ (for this instance) then the session key is determined by the Simulator as above, otherwise the session key is set to a random and independent value.

Corruption On receiving a $\mathsf{Corrupt}$ call from $\mathcal{Z}$ for instance P_i in session ssid, the simulator $\mathcal{S}$ calls the $\mathsf{Corrupt}$ routine of $\widehat{\mathcal{F}}_{\text{PAKE}}$ to obtain pwd. If $\mathcal{S}$ had already output a message to $\mathcal{Z}$, and not output sk_1 (via a call to NewKey) it computes

$$W_1 = \mathsf{sim}(\textsc{crs}, \tau, \langle R_1, S_1/\mathrm{pwd}, T_1, \iota_1 \rangle).$$

and outputs this W_1 along with pwd, and $\textsc{hk}_1$. as internal state of P_i. Note that this computation of W_1 is identical to the computation of W_1 in the computation of key above used to call NewKey (which is really output to $\mathcal{Z}$ only when $\mathrm{pwd}' = \mathrm{pwd}$).

Without loss of generality, we can assume that in the real-world if the Adversary (or Environment $\mathcal{Z}$) corrupts an instance before the session key is output then the instance does not output any session key. This is so because the Adversary (or $\mathcal{Z}$) either sets the key for that session or can compute it from the internal state it broke into.

Proof of Indistinguishability - Series of Experiments. We now describe a series of experiments between a probabilistic polynomial time challenger $\mathcal{C}$ and the environment $\mathcal{Z}$, starting with Expt_0 which we describe next. We will show that the view of $\mathcal{Z}$ in Expt_0 is same as its view in UC-PAKE ideal-world setting with $\mathcal{Z}$ interacting with $\widehat{\mathcal{F}}_{\text{PAKE}}$ and the UC-PAKE simulator $\mathcal{S}$ described above. We end with an experiment which is identical to the real world execution of the protocol in Fig 4. We will show that the environment has negligible advantage in distinguishing between these series of experiments, leading to a proof of realization of $\mathcal{F}_{\text{PAKE}}$ by the protocol Π.

Here is the complete code in Expt_0 (stated as it's overall experiment with $\mathcal{Z}$):

1. The challenger $\mathcal{C}$ picks the CRS just as in the real world, except the QA-NIZK $\textsc{crs}$ is generated using the crs-simulator crssim, which also generates a simulator trapdoor τ. $\mathcal{C}$ retains a, τ, hk.
2. On receiving $\mathtt{NewSession}, sid, \mathsf{ssid}, P_i, P_j, \mathrm{pwd}, role$ from $\mathcal{Z}, \mathcal{C}$ generates ($\textsc{hk}_1$, $\textsc{hp}_1$) by running $\mathsf{ver.hkgen}(\textsc{crs})$. Next, it generates R_1, S_1, T_1 by choosing r_1, r_1', r_1'' at random, and setting $R_1 = \mathbf{g}_1^{r_1}$, $S_1 = \mu \cdot \mathbf{a}^{r_1} \cdot \mathbf{g}_1^{r_1'}$, $T_1 = \mathbf{g}_1^{r_1''}$. It sends these values along with $\textsc{hp}_1$ to $\mathcal{Z}$.
3. On receiving $R_2', S_2', T_2', \textsc{hp}_2'$ from $\mathcal{Z}$, intended for session ssid and party P_i (and assuming no corruption of this instance)
   (a) if the received elements are either not in their respective groups, or are trivially 1, output sk_1 chosen randomly and independently from $\mathbb{G}_T$.
   (b) Otherwise, if the message received is identical to message sent by $\mathcal{C}$ in the same session (i.e. same ssid) on behalf of the peer, then output $\mathrm{sk}_1 \xleftarrow{\$} \mathbb{G}_T$ (unless the simulation of peer also received a legitimate message and its key has already been set, in which case the same key is used to output sk_1 here).
   (c) Else, compute $\mathrm{pwd}' = S_2'/(R_2')^a$. If $\mathrm{pwd}' \neq \mathrm{pwd}$ (note pwd was given in $\mathtt{NewSession}$ request), then output sk_1 randomly and independently from $\mathbb{G}_T$.

(d) Else, compute $\iota_2' = \mathcal{H}(sid, \mathsf{ssid},\ P_j, P_i,\ R_2', S_2', \textsc{hp}_2')$.
if $T_2' \neq \mathsf{sphf.privH}(\mathsf{hk}, \langle R_2', S_2'/\mathrm{pwd}\rangle, \iota_2')$ then output a random value in $\mathbb{G}_T$.
Else, compute $W_1 = \mathsf{sim}(\textsc{crs},\ \tau,\ \langle R_1, S_1/\mathrm{pwd}, T_1, \iota_1\rangle)$, where $\iota_1 = \mathcal{H}(sid,$ $\mathsf{ssid},\ P_i, P_j,\ R_1, S_1, \textsc{hp}_1)$, and output

$$\mathsf{ver.privH}(\textsc{hk}_1, \langle R_2', S_2'/\mathrm{pwd}, T_2, \iota_2'\rangle) \cdot \mathsf{ver.pubH}(\textsc{hp}_2', W_1).$$

4. On a $\mathsf{Corrupt}$ call, if Step 2 has already happened then output $\textsc{hk}_1$, pwd and $W_1 = \mathsf{sim}(\textsc{crs}, \tau, \langle R_1, S_1/\mathrm{pwd}, T_1, \iota_1\rangle)$,

All outputs of sk_1 are also accompanied with sid, ssid (but are not mentioned above for ease of exposition).

Note that each instance has two asynchronous phases: a phase in which $\mathcal{C}$ outputs $R_1, S_1, ...$ to $\mathcal{Z}$, and a phase where it receives a message from $\mathcal{Z}$. However, $\mathcal{C}$ cannot output sk_1 until it has completed both phases. These orderings are dictated by $\mathcal{Z}$. We will consider two different kinds of temporal orderings. A temporal ordering of different instances based on the order in which $\mathcal{C}$ outputs sk_1 in an instance will be called **temporal ordering by key output**. A temporal ordering of different instances based on the order in which $\mathcal{C}$ outputs its first message (i.e. $R_1, S_1, ...$) will be called **temporal ordering by message output**. It is easy to see that $\mathcal{C}$ can dynamically compute both these orderings by maintaining a counter (for each ordering).

It is straightforward to inspect that the view of $\mathcal{Z}$ in Expt_0 is identical to its view in its combined interaction with $\widehat{\mathcal{F}}_{\textsc{pake}}$ and $\mathcal{S}$, as $\mathcal{C}$ has just combined the code of $\widehat{\mathcal{F}}_{\textsc{pake}}$ and $\mathcal{S}$ (noting that in Step 3(d), pwd = pwd$'$)

Expt_1 : In this experiment Step 3(c) is dropped altogether and Step 3(d) altered as follows: In Step 3(d) in Expt_0, the condition $T_2' \neq \mathsf{sphf.privH}(\mathsf{hk}, \langle R_2', S_2'/\mathrm{pwd}\rangle, \iota_2')$ is replaced by "if $(S_2' \neq \mathrm{pwd} \cdot (R_2')^a)$ or $T_2' \neq \mathsf{sphf.privH}(\mathsf{hk}, \langle R_2', S_2'/\mathrm{pwd}\rangle, \iota_2')$". Rest of the computation of sk_1 in Step 3(d) remains the same.

We claim that the view of $\mathcal{Z}$ is statistically identical in Expt_0 and Expt_1. This follows by noting that $S_2' \neq \mathrm{pwd} \cdot (R_2')^a$ is equivalent to the condition $\mathrm{pwd}' \neq \mathrm{pwd}$ in Expt_0. The condition $S_2' = \mathrm{pwd} \cdot (R_2')^a$ held in Step 3(d) in Expt_0, as that step was only reached if this condition held.

Expt_2 : In this experiment, in Step 3(d) the condition is replaced by just "if $T_2' \neq$ $\mathsf{sphf.privH}(\mathsf{hk}, \langle R_2', S_2'/\mathrm{pwd}\rangle, \iota_2')$", i.e. the disjunct $(S_2' \neq \mathrm{pwd} \cdot (R_2')^a)$ is dropped.

First note that T_1 is being computed randomly. The experiment Expt_2 is then statistically indistinguishable from Expt_1 by smoothness of sphf (note that it can be shown that the polynomial number of extra bits of information leaked by the conditions $T_2' \neq \mathsf{sphf.privH}(\mathsf{hk}, \langle R_2', S_2'/\mathrm{pwd}\rangle, \iota_2')$ themselves have a negligible effect on the smoothness of the sphf – this argument is employed in the Cramer-Shoup CCA2-encryption scheme [CS02]).

Correcting Message Outputs to use pwd

Expt_3 : In this experiment the challenger in Step 2 computes S_1 in each instance as $\mathrm{pwd} \cdot \mathbf{a}^{r_1} \cdot \mathbf{g}_1^{r'_1}$. Note the use of pwd instead of μ.

This is statistically the same, as in each instance the challenger picks a fresh and random r'_1, and it is not used anywhere else.

Expt_4 : In each instance, S_1 is computed as follows: $\mathrm{pwd}\cdot\mathbf{a}^{r_1}$. Further, T_1 is computed as follows: $T_1 = \mathsf{sphf.pubH}(\mathsf{hp}, \langle R_1, S_1/\mathrm{pwd}\rangle, \iota_1)$.

To show that Expt_3 is computationally indistinguishable from Expt_4, we define several hybrid experiments $\mathsf{Expt}_{3,i}$ inductively. Experiment $\mathsf{Expt}_{3,0}$ is identical to Expt_3. If there are a total of N instances, $\mathsf{Expt}_{3,N}$ will be identical to Expt_4. Experiment $\mathsf{Expt}_{3,i+1}$ differs from experiment $\mathsf{Expt}_{3,i}$ in only (temporally ordered by message output) the $(i{+}1)$-th instance. While in $\mathsf{Expt}_{3,i}$, the $(i{+}1)$-th instance is simulated by $\mathcal{C}$ as in Expt_3, in $\mathsf{Expt}_{3,i+1}$ this instance is simulated as in Expt_4.

Lemma 3. *For all $i : 0 \le i \le N$, the view of $\mathcal{Z}$ in experiment $\mathsf{Expt}_{3,i+1}$ is computationally indistinguishable from the view of $\mathcal{Z}$ in $\mathsf{Expt}_{3,i}$.*

Proof. We define several hybrid experiments. Experiment $\mathbf{G}_0$ is identical to $\mathsf{Expt}_{3,i}$.

In $\mathbf{G}_1$, in the $(i+1)$-th instance T_1 is computed differently:

$$T_1 = \mathsf{sphf.privH}(\mathsf{hk}, \langle R_1, S_1/\mathrm{pwd}\rangle, \iota_1) \tag{2}$$

This is statistically the same as all other T_1 are either randomly computed (in instances greater than $(i+1)$), or are computed using the public hash with hp (in instances less than $(i+1)$). Then the claim follows by smoothness of sphf, and noting that $R_1^a \neq S_1/\mathrm{pwd}$ in instance $(i+1)$ (by construction of $\mathsf{Expt}_{3,i}$).

In the next experiment $\mathbf{G}_2$, the challenger generates the S_1 in the $(i+1)$-th instance as follows: $S_1 = \mathrm{pwd} \cdot \mathbf{a}^{r_1}$. That the view of $\mathcal{Z}$ in experiments $\mathbf{G}_1$ and $\mathbf{G}_2$ are computationally indistinguishable follows from the DDH assumption in group $\mathbb{G}_1$ (note a is not being used by the challenger, now that Step 3(c) is no more).

In the next experiment $\mathbf{G}_3$, change the computation of T_1 in session $(i+1)$ to use the public hash (of sphf) and witness r_1. Since, now R_1 and S_1/pwd are in the Diffie Hellman language, indistinguishabilty from the previous experiment follows by correctness of sphf. □

Expt_5 : In this experiment, the CRS is generated using crsgen instead of the crs-simulator, and W_1 is computed everywhere by prover of the QA-NIZK instead of the proof simulator.

Indistinguishability from the previous experiment follows by zero-knowledge property of the QA-NIZK, noting that all proofs being generated are on language memebers.

At this point, the complete experiment Expt_5 can be described as follows:

1. The challenger $\mathcal{C}$ Picks the CRS just as in the real world. It retains a, hk.
2. On receiving $\texttt{NewSession}, sid, \mathsf{ssid}, P_i, P_j, \mathrm{pwd}, role$ from $\mathcal{Z}, \mathcal{C}$ generates ($\textsc{hk}_1$, $\textsc{hp}_1$) by running $\mathsf{ver.hkgen}(\textsc{crs})$. Next, it generates R_1, S_1, T_1 by choosing r_1 at random, and setting $R_1 = \mathbf{g}_1^{r_1}$, $S_1 = \mathrm{pwd} \cdot \mathbf{a}^{r_1}$, $T_1 = \mathsf{sphf.pubH}(\mathsf{hp}, \langle R_1, S_1/\mathrm{pwd}\rangle, \iota_1)$, where $\iota_1 = \mathcal{H}(sid, \mathsf{ssid}, P_i, P_j, R_1, S_1, \textsc{hp}_1)$. It sends these values along with $\textsc{hp}_1$ to $\mathcal{Z}$.
3. On receiving $R_2', S_2', T_2', \textsc{hp}_2'$ from $\mathcal{Z}$, intended for session ssid and party P_i (and assuming no corruption of this instance)
   (a) if the received elements are either not in their respective groups, or are trivially 1, output sk_1 chosen randomly and independently from $\mathbb{G}_T$.
   (b) Otherwise, if the message received is identical to message sent by $\mathcal{C}$ in the same session (i.e. same ssid) on behalf of the peer, then output $\mathrm{sk}_1 \xleftarrow{\$} \mathbb{G}_T$ (unless the simulation of peer also received a legitimate message and its key has already been set, in which case the same key is used to output sk_1 here).
   (c) -
   (d) Else, compute $\iota_2' = \mathcal{H}(sid, \mathsf{ssid}, P_j, P_i, R_2', S_2', \textsc{hp}_2')$.
   if $T_2' \neq \mathsf{sphf.privH}(\mathsf{hk}, \langle R_2', S_2'/\mathrm{pwd}\rangle, \iota_2')$ then output randomly from $\mathbb{G}_T$.
   (e) Else, Compute $W_1 = \mathsf{prover}(\textsc{crs}, \langle R_1, S_1/\mathrm{pwd}, T_1, \iota_1\rangle; r_1)$. Output
   $$\mathsf{ver.privH}(\textsc{hk}_1, \langle R_2', S_2'/\mathrm{pwd}, T_2, \iota_2'\rangle) \cdot \mathsf{ver.pubH}(\textsc{hp}_2', W_1).$$
4. On a $\mathsf{Corrupt}$ call, if Step 2 has already happened then output $\textsc{hk}_1$, pwd and $W_1 = \mathsf{prover}(\textsc{crs}, \langle R_1, S_1/\mathrm{pwd}, T_1, \iota_1\rangle; r_1)$,

Handling Legitimate Messages

Expt_6 : In this experiment the Step 3(b) is modified as follows:
Step 3(b): Otherwise, if the message received is identical to message sent by $\mathcal{C}$ in the same session (i.e. same SSID) on behalf of the peer, **and** if simulation of peer also received a legitimate message and its key has already been set, then output that same key here. Else, go to Step 3(e).

To show that Expt_6 is indistinguishable from Expt_5 we need to go through several hybrid experiments. In each subsequent hybrid experiment one more instance is modified, and the order in which these instances are handled is determined by temporal order of key output. In the hybrid experiment $\mathsf{Expt}_{5,i}$ ($N \geq i \geq 1$), the Step 3(b) in the i-th temporally ordered instance is modified as required in Expt_6 description above. Experiment $\mathsf{Expt}_{5,0}$ is same as experiment Expt_5, and experiment $\mathsf{Expt}_{5,N}$ is same as experiment Expt_6.

Lemma 4. *For all $i \in [1..N]$, experiment $\mathsf{Expt}_{5,i}$ is computationally indistinguishable from $\mathsf{Expt}_{5,i-1}$.*

Proof. The lemma is proved using several hybrid experiments of its own. The experiment $\mathbf{H}_0$ is same as $\mathsf{Expt}_{5,i-1}$.

In experiment $\mathbf{H}_1$ the CRS is set as in the real world, except that the QA-NIZK CRS is set using the crs simulator crssim (the challenger retains the trapdoor η output by the crs simulator). All proofs W_1 are still computed using the prover. Experiments $\mathbf{H}_0$ and $\mathbf{H}_1$ are indistinguishable as the QA-NIZK has the property that the simulated CRS and the real-world CRS are statistically identical.

In experiment $\mathbf{H}_2$, in instance i, the value W_1 (in Step 3(e) or corruption) is generated using the proof simulator using trapdoor η. Indistinguishability follows by zero-knowledge property of the QA-NIZK as the proof being generated is on a language member.

In experiment $\mathbf{H}_3$, in instance i, the value T_1 is generated using the private hash key hk, and the private hash function $\mathsf{sphf.privH}$ (thus eliminating the use of witness r_1). Experiments $\mathbf{H}_2$ and $\mathbf{H}_3$ are indistinguishable by the correctness of sphf.

In experiment $\mathbf{H}_4$, in instance i, the values R_1, S_1 are generated as $R_1 = \mathbf{g}_1^{r_1}$, $S_1 = \text{pwd} \cdot \mathbf{a}^{r_1} \cdot \mathbf{g}_1^{r_1'}$. where r_1, r_1' are random and independent. This follows by employing DDH on $\mathbf{g}_1, \mathbf{g}_1^{r_1}, \mathbf{a}$ and either $\mathbf{g}_1^{ar_1}$ or $\mathbf{g}_1^{ar_1+r_1'}$.

In experiment $\mathbf{H}_5$, in peer of instance i, in Step 3(d) the condition $T_2' \neq \mathsf{sphf.privH}(\mathsf{hk}, \langle R_2', S_2'/\text{pwd}\rangle, \iota_2')$ is replaced by "if $(S_2' \neq \text{pwd} \cdot (R_2')^a)$ or $T_2' \neq \mathsf{sphf.privH}(\mathsf{hk}, \langle R_2', S_2'/\text{pwd}\rangle, \iota_2')$". Indistinguishability from experiment $\mathbf{H}_4$ follows by smooth$_2$ property of the sphf, noting that at most one bad $\mathsf{sphf.privH}$ is being output to the Adversary (namely T_1 in instance i).

In experiment $\mathbf{H}_6$, in instance i, change Step 3(b) as follows: Step 3(b): Otherwise, if the message received is identical to message sent by $\mathcal{C}$ in the same instance (i.e. same SSID) on behalf of the peer,

- if simulation of peer also received a legitimate message and its key has already been set, then output that same key here. If peer is corrupted, output the key supplied by the Adversary.
- Else, compute $\iota_2' = \mathcal{H}(sid, \mathsf{ssid}, P_j, P_i, R_2', S_2', \mathrm{HP}_2')$, Output

$$\mathsf{ver.privH}(\mathrm{HK}_1, \langle R_2', S_2'/\text{pwd}, T_2'\rangle, \iota_2') \cdot \mathsf{ver.privH}(\mathrm{HK}_2, \langle R_1, S_1/\text{pwd}, T_1\rangle, \iota_1)$$

Here HK_2 is the HK output by $\mathsf{ver.hkgen}$ in the peer instance of instance i.

The experiments $\mathbf{H}_6$ and $\mathbf{H}_5$ are computationally indistinguishable by noting the following three facts:

1. In the peer of instance of instance i (which generated HK_2), in Step 3(e) the computation $\mathsf{ver.privH}(\mathrm{HK}_2, \cdot)$ is on a language member, as Step 3(e) is only reached if the condition in Step 3(d) is false (which implies language membership of the incoming tuple).
2. Also, note that only one QA-NIZK proof is being simulated and that is in this same instance, but in a mutually exclusive step (Step 3(e) or corruption). Moreover, the CRS generated by the crs simulator is statistically identical to the CRS geenrated by crsgen.
3. Then, $\mathsf{ver.privH}(\mathrm{HK}_2, \langle R_1, S_1/\text{pwd}, T_1\rangle, \iota_1)$ is random even when the adversary is given HP_2 by smoothness of the QA-NIZK, since $S_1/\text{pwd} \neq R_1^a$.

In experiment $\mathbf{H}_7$, in peer of instance i, in Step 3(d) the condition "if $(S'_2 \neq \mathrm{pwd} \cdot (R'_2)^a)$ or $T'_2 \neq \mathsf{sphf.privH}(\mathsf{hk}, \langle R'_2, S'_2/\mathrm{pwd}\rangle, \iota'_2)$" is replaced by "if $T'_2 \neq \mathsf{sphf.privH}(\mathsf{hk}, \langle R'_2, S'_2/\mathrm{pwd}\rangle, \iota'_2)$". Indistinguishability from experiment $\mathbf{H}_6$ follows by smooth$_2$ property of the sphf, noting that at most one bad $\mathsf{sphf.privH}$ is being output to the Adversary (namely T_1 in instance i).

In experiment $\mathbf{H}_8$, in instance i, R_1, S_1 are generated as $R_1 = \mathbf{g}^{r_1}$, $S_1 = \mathrm{pwd} \cdot \mathbf{a}^{r_1}$, by employing DDH.

In experiment $\mathbf{H}_9$, in instance i, T_1 is generated using the public hash key hp, and witness r_1. Indistinguishability follows by correctness of the sphf.

In experiment $\mathbf{H}_{10}$, the QA-NIZK is generated using the real world CRS generator. Moreover, in instance i, in Step 3(e) and corruption step, W_1 is computed using the real world prover. Indistinguishability follows by zero-knowledge property of the QA-NIZK.

In experiment $\mathbf{H}_{11}$, in Step 3(b) the key is output as follows:

– Else, compute $\iota'_2 = \mathcal{H}(sid, \mathsf{ssid}, P_j, P_i, R'_2, S'_2, \textsc{hp}'_2)$.
  Compute $W_1 = \mathsf{prover}(\textsc{crs}, \langle R_1, S_1/pwd, T_1, \iota_1\rangle, r_1)$. Output

$$\mathsf{ver.privH}(\textsc{hk}_1, \langle R'_2, S'_2/\mathrm{pwd}, T'_2\rangle, \iota'_2) \cdot \mathsf{ver.pubH}(\textsc{hp}'_2, W_1)$$

Indistinguishability follows by noting that $\textsc{hp}'_2$ is exactly the $\textsc{hp}_2$ computed by the challenger in the peer instance. The claim then follows by completeness of the smooth QA-NIZK.

The induction step is complete now, as the above computation of the session key is same as in Step 3(e). □

Handling Adversarial Messages

Expt_7 : In this experiment in Step 3(d) the condition is changed to "if $(S'_2 \neq \mathrm{pwd} \cdot (R'_2)^a)$ or $T'_2 \neq \mathsf{sphf.privH}(\mathsf{hk}, \langle R'_2, S'_2/\mathrm{pwd}\rangle, \iota'_2)$". In other words, the disjunct $(S'_2 \neq \mathrm{pwd} \cdot (R'_2)^a)$ is introduced.

Indistinguishability follows by the same argument as employed in experiments Expt_2 and Expt_1.

Expt_8 : In this experiment Step 3(d) is dropped altogether.

We first show that if $(S'_2 \neq \mathrm{pwd} \cdot (R'_2)^a)$ or $T'_2 \neq \mathsf{sphf.privH}(\mathsf{hk}, \langle R'_2, S'_2/\mathrm{pwd}\rangle, \iota'_2)$, then $R'_2, S'_2/\mathrm{pwd}, T'_2$ and ι'_2 are not in language L (for which the QA-NIZK is defined). Clearly, if the first disjunct does not hold then the tuple is not in the language. So, suppose $S'_2 = \mathrm{pwd} \cdot (R'_2)^a$, with witness r_2 for R'_2. Then, by correctness of the sphf,

$$\mathsf{sphf.privH}(\mathsf{hk}, \langle R'_2, S'_2/\mathrm{pwd}\rangle, \iota'_2) = \mathsf{sphf.pubH}(\mathsf{hp}, \langle R'_2, S'_2/\mathrm{pwd}\rangle, \iota'_2; r_2).$$

Then again, the tuple is not in the language.

Thus, $\mathsf{ver.privH}(\textsc{hk}_1, \langle R'_2, S'_2/\mathrm{pwd}, T'_2\rangle, \iota'_2)$ is random, even when the Adversary is given $\textsc{hp}_1$, by smooth-soundness of the QA-NIZK.

Expt_9 : In this experiment the Step 3(b) is dropped. In other words, the challenger code goes straight from 3(a) to 3(e).

Experiments Expt_9 and Expt_8 produce the same view for $\mathcal{Z}$, since if both peers (of a instance) received legitimate messages forwarded by $\mathcal{Z}$, then Step 3(e) computes the same instance key in both instances.

Finally, a simple examination shows that the view of $\mathcal{Z}$ in Expt_9 is identical to the real world protocol.