Smooth NIZK Arguments with Applications to Asymmetric UC-PAKE

Charanjit S. Jutla and Arnab Roy

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 SPH 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 ($F_{pake}$). Gentry et al [GMR06] extended the functionality of symmetric UC-PAKE [CHK+05] to the asymmetric setting ($F_{apwKE}$) 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 $F_{apwKE}$ to have additional TestAbort functions.

Our Contributions. In this paper, we give the first single-round UC-APAKE protocol (realizing $F_{apwKE}$). In fact, both parties just send a single message asynchronously. Since this is a single round protocol, we can realize $F_{apwKE}$ 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 num-
ber 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] con-
struction 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 def-
initional 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 adversar-
ially inserted, the simulator must have the capability to extract password $pw'$
from it, so that it can feed the ideal functionality $F_{apwKE}$ 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 $n \times m \times \text{poly}(q)$, 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. [FLR10]. 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 $F_{\text{apKE}}$ for asymmetric password-authenticated key-exchange, construct the new single-round UC-APAKE protocol and provide its proof of UC-realization of ideal functionality $F_{\text{apKE}}$.

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 $\{L_{\rho}\}_{\rho \in \text{Lpar}}$ with the parameters coming from an associated parameter language $\text{Lpar}$. An SPHF consists of the following efficient algorithms.

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

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

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

A projective hash function family is called smooth if for all $x \notin L_{\rho}$, $\text{privH}(\text{hk}, x, l)$ is statistically indistinguishable from a random element in $\Pi$, even given $\text{hp}$. It is called smooth$_2$ if for all $x \notin L_{\rho}$, $\text{privH}(\text{hk}, x, l)$ is statistically indistinguishable from a random element in $\Pi$, even given $\text{hp}$ and one evaluation of $\text{privH}(\text{hk}, x^*, l^*)$ 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 $L_\rho$). 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 $(\text{pargen, crsgen, prover, ver})$ a (labeled) quasi-adaptive non-interactive zero-knowledge (QA-NIZK) proof system for witness-relations $\mathcal{R}_\lambda = \{R_\lambda\}$ with parameters sampled from a distribution $\mathcal{D}$ over associated parameter language $L_{\text{par}}$, if there exist efficient simulators $\text{crs}\text{sim}, \text{ver}\text{sim}$ such that for all non-uniform PPT adversaries $A_1, A_2, A_3$ we have (in all of the following probabilistic experiments, the experiment starts by setting $\lambda$ as $\lambda \leftarrow \text{pargen}(1^m)$, and choosing $\rho$ as $\rho \leftarrow \mathcal{D}_\lambda$):

**Quasi-Adaptive Completeness:**

\[
\Pr \left[ \text{crs} \leftarrow \text{crsgen}(\lambda, \rho); (x, w) \leftarrow A_1(\text{crs}, \rho); \pi \leftarrow \text{prover}(\text{crs}, x, w); \right. \\
\left. \text{ver}(\text{crs}, x, \pi) = 1 \text{ if } R_\rho(x, w) \right] = 1
\]

**Quasi-Adaptive (Computational) Soundness:**

\[
\Pr[\text{crs} \leftarrow \text{crsgen}(\lambda, \rho); (x, \pi) \leftarrow A_2(\text{crs}, \rho); x \notin L_\rho \land \text{ver}(\text{crs}, x, \pi) = 1] \approx 0
\]

**Quasi-Adaptive Zero-Knowledge:**

\[
\Pr[\text{crs} \leftarrow \text{crsgen}(\lambda, \rho); (x, w) \leftarrow A_3^{\text{prover}(\text{crs}, \pi)}(\text{crs}, \rho); \pi \leftarrow \text{prover}(\text{crs}, x, w); \\
\left. \text{ver}(\text{crs}, x, \pi) = 1 \text{ if } R_\rho(x, w) \right] \approx 0
\]

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

The above completeness property is now defined as:

\[
\Pr \left[ \text{crs} \leftarrow \text{crsgen}(\lambda, \rho); (x, w) \leftarrow A_1(\text{crs}, \rho); \pi \leftarrow \text{prover}(\text{crs}, x, w); \\
\left. (\text{hp}, \text{hk}) \leftarrow \text{hkgen(\text{crs})}; \text{privH}(\text{hk}, x) = \text{pubH}(\text{hp}, \pi) \text{ if } R_\rho(x, w) \right] = 1
\]

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

More precisely, Quasi-Adaptive Smooth-Soundness is the following property (let $\mathcal{U}$ be the uniform distribution on the range of $\text{privH}$, which is assumed to
be of cardinality exponential in $m$): for every two-stage efficient oracle adversary $A$
\[
\text{crs} \leftarrow \text{crs} \text{gen}(\lambda, \rho); (\text{hp}, \text{hk}) \leftarrow \text{hk} \text{gen}(\text{crs}); (x^*, \pi^*, \sigma) \leftarrow \mathcal{A}^O(\text{crs}, \rho, \text{hp}); u \leftarrow \mathcal{U}; \\
\Pr[\mathcal{A}^O(\text{priv} \text{H}(\text{hk}, x^*), \sigma) = 1 \mid Q] - \Pr[\mathcal{A}^O(u, \sigma) = 1 \mid Q] \approx 0
\]
where the oracle $O$ is instantiated with $\text{priv} \text{H}(\text{hk}, \cdot)$, and $Q$ is the condition that $x^*$ is not in the language $L_\rho$ and all oracle calls by the adversary in both stages are with $L_\rho$-language members. Here, $\sigma$ is a local state of $A$.

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

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

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 $\text{sphf}$ and the QA-NIZK verifier functionalities with the keyword $\text{ver}$.

## 4 Matrix Decisional Assumptions

We will consider bilinear groups that consist of three cyclic groups of prime order $q$, $G_1$, $G_2$ and $G_T$ with an efficient bilinear map $e : G_1 \times G_2 \rightarrow G_T$. Group elements $g_1$ and $g_2$ will typically denote generators of the group $G_1$ and $G_2$ respectively. Following [EHK+13], in this section and the next we will use the notations $[a]_1$, $[a]_2$ and $[a]_T$ to denote $a g_1$, $a g_2$, and $a \cdot e(g_1, g_2)$ respectively and use additive notations for group operations. When talking about a general group $G$ with generator $g$, we will just use the notation $[a]$ to denote $ag$. 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 $D_{l,k}$, 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 $D_{l,k}$-MDDH assumption in group $G$ states that with samples $A \leftarrow D_{l,k}, s \leftarrow \mathbb{Z}_q^k, s' \leftarrow \mathbb{Z}_q^l$, the tuple $([A], [As])$ is computationally indistinguishable from $([A], [s'])$. A matrix distribution $D_{k+1,k}$ is simply denoted by $D_k$.

Intuitively, a $D_{l,k}$-MDDH assumption allows us to generate $l$ (computationally) independently random group elements from an initial $k$ independently random exponents. A $D_k$-MDDH assumption allows us to generate one extra random group element. In this section, we will establish that, in fact, a $D_k$-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 $D_{l,k}$-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 $\overline{A}$ to be the top $k \times k$ square sub-matrix of $A$ and $\underline{A}$ to be the bottom $(l - k) \times k$ sub-matrix of $A$.

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

- For $A \leftarrow D_k, B \leftarrow D_{l,k}, E \leftarrow E, F \leftarrow F$, we have:
  
  $B \approx \overline{A}, \quad \underline{B} \approx \overline{A} \approx FA$.

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

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

Proof of this theorem can be found in Appendix A.

**Corollary 1.** Any $D_k$ distribution can be boosted to a $D_{l,k}$ 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 $D_{l,k}$ sample to be the top $k \times k$ matrix of a $D_k$ sample and setting the bottom $(l - k) \times k$ sub-matrix of the $D_{l,k}$ sample to be uniformly random in $\mathbb{Z}^{(l-k)\times k}$. The required distributions $E$ and $F$ 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 $D_k$ 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 $\text{ver.hkgen}$ 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 $[M]_1$ as below:

$$L_{[M],1} = \{ [M]_1 \mathbf{x} \in \mathbb{G}_1^n \mid \mathbf{x} \in \mathbb{Z}_q^n \},$$

where $[M]_1$ is an $n \times t$ matrix of $\mathbb{G}_1$ elements.

Here $[M]_1$ is an element of the associated parameter language $L_{\text{par}}$, which is all $n \times t$ matrices of $\mathbb{G}_1$ elements. The parameter language $L_{\text{par}}$ also has a corresponding witness relation $R_{\text{par}}$, where the witness is a matrix of $\mathbb{Z}_q$ elements: $R_{\text{par}}([M]_1, M)$ iff $[M]_1 = M \cdot g_1$.

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

**Smooth QA-NIZK Construction.** We now describe a smooth computationally-sound Quasi-Adaptive NIZK ($K_0, K_1, P, V$) for linear subspace languages $\{L_{[M],1}\}$ with parameters sampled from a robust and efficiently witness-samplable distribution $D$ over the associated parameter language $L_{\text{par}}$ and given a $D_k$-MDDH assumption.

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

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

$$\text{CRS}_p := ([P]_1 = [M^\top K]_1)$$

$$\text{CRS}_v := ([C]_2^{k \times k} = [K A]_2, \ [A]_2^{k \times k})$$

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

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

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

$$hk := [C]_2 s, \quad hp := [A]_2 s$$

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

$$\text{privH}(hk, [y]_1) := e([y]_1^\top, hk), \quad \text{pubH}(hp, \pi) := e(\pi, hp)$$
Theorem 2. The above algorithms \( \{K_0, K_1, P, V\} \) constitute a smooth computationally-abstract Quasi-Adaptive NIZK proof system for linear subspace languages \( \{L[M]\} \) with parameters \( [M]_1 \) sampled from a robust and efficiently witness-samplable distribution \( D \) over the associated parameter language \( L\text{par} \), given any group generation algorithm for which the \( D_k\text{-MDDH} \) assumption holds for group \( G_2 \).

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 \( \text{privH} \) 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 \( G_0 \) just replicates the construction, but samples \( A \) from a distribution \( D_{k+n-t,k} \) obtained by boosting the given distribution \( D_k \) by Corollary 1. The construction only uses the top \( k \times k \) sub-matrix \( A \) of the sample which is distributed identically for both \( D_k \) and \( D_{k+n-t,k} \). Let \( A \) be the bottom \((n-t) \times k\) sub-matrix of \( A \).

In Game \( G_1 \), the challenger efficiently samples \( [M]_1 \) according to distribution \( D \), along with witness \( M \) (since \( D \) is an efficiently witness samplable distribution). Since \( M \) is an \( n \times t \) dimensional rank \( t \) matrix, there is a rank \( n-t \) matrix \( M^\perp \) of dimension \( n \times (n-t) \) whose columns form a complete basis for the kernel of \( M^\top \), which means \( M^\top M^\perp = 0^{t \times (n-t)} \). In this game, the NIZK CRS is computed as follows: Generate matrix \( K' \) uniformly randomly from \( \mathbb{Z}_q^{n \times k} \) and compute the matrix \( T^{(n-t) 	imes k} \), such that \( TA = A \). Implicitly set: \( K = K' + M^\perp T \). Therefore we have,

\[
\text{CRS}_p^{n \times k} = \left[M^\top K\right]_1 = \left[M^\top (K' + M^\perp T)\right]_1 = \left[M^\top K'\right]_1
\]

\[
\left[C\right]_2^{n \times k} = \left[(K' + M^\perp T)A\right]_2 = K'[A]_2 + M^\perp [A]_2,
\]

\[
hk = [C]_2 s, \quad hp = [A]_2 s
\]

In Game \( G_2 \), we sample a fresh random vector \( s' \) in \( \mathbb{Z}_q^{n-t} \) and modify the simulated computations as follows:

\[
\text{CRS}_p^{n \times k} = \left[M^\top K'\right]_1, \quad \left[C\right]_2^{n \times k} = K'[A]_2 + M^\perp [A]_2,
\]

\[
hk = K'[\hat{As}]_2 + M^\perp [s']_2, \quad hp = [\hat{As}]_2
\]

Given a \( D_{k+n-t,k} \) challenge which is either “real”: \( ([A]_2, [\hat{As}]_2, [As]_2) \) or “fake”: \( ([A]_2, [\hat{As}]_2, [s']_2) \), we observe that the real tuple can be used to simulate Game \( G_1 \), while the fake tuple can be used to simulate Game \( G_2 \). Thus the games \( G_1 \) and \( G_2 \) are indistinguishable by the \( D_{k+n-t,k}\text{-MDDH} \) assumption, which in turn is implied by the \( D_k\text{-MDDH} \) assumption by Theorem 1.

Now in Game \( G_2 \) we have,

\[
\text{privH}(hk, [y^*]_1) = e \left( [y^*]_1^\top, K'[\hat{As}]_2 + M^\perp [s']_2 \right)
\]
For the oracle queries where $\{y^*\}_1 \in L_{\{M\}_1}$, we have $y^{*\top}M_1 = 0^{1 \times (n-t)}$. Hence the simulator responds with $e([y^*]_1^\top, K'[\overline{A}_s]_2)$. Note that $s'$ does not appear in this response.

For the adversary supplied $\{y^*\}_1 \not\in L_{\{M\}_1}$, we have $y^{*\top}M_1 \neq 0^{1 \times (n-t)}$. Therefore $\text{privH}(hk, y^*)$ is uniformly random, as $s'$ 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 \text{TestPwd} capability available to the ideal world adversary: if \text{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 \text{OfflineTestPwd} tests to mimic a similar capability in the real world (in fact, the ideal world adversary is even allowed to perform \text{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 $\mathcal{F}_{\text{APKE}}$ 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 $\mathcal{F}_{\text{APKE}}$ is described in detail in Fig. 1.
proofs. The essential idea of [BP13] is that while the Client holds the actual password data, an adversary may intercept and store password data records. Upon receiving a query, the functionality [SpwKE] may store password data records (file, P_i, P_j, pw) and mark it un compromised. If there is a password data record (file, P_i, P_j, pw) that is marked compromised, then store (file, P_i, P_j, pw) and mark it fresh.

Stealing Password Data

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

Upon receiving a query (OfflineTestPwd, sid, pw') from Adversary S:
If there is a password data record or if there is a password data record (file, P_i, P_j, pw) that is marked un compromised, then store (offline, pw'). Otherwise do: if pw = pw', send pw back to S. If pw ≠ pw', reply with 'wrong guess'.

Active Session Attacks

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

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

Key Generation and Authentication

Upon receiving a query (NewKey, sid, sid, P_i, sk) from S, where |sk| = k:
If there is a session record of the form (role, sid, P_i, P_j, pw) that is not marked completed,
- If this record is compromised, or either P_i or P_j is corrupted, then output (sid, sid, sk) to player P_i.
- If this record is fresh, and there is a session record (role, sid, P_j, P_i, pw') with pw' = pw, and a key sk' was sent to P_i, and (role, sid, P_i, P_j, pw) was fresh at the time, then output (sid, sid, sk') to P_i.
- In any other case, pick a new random key sk' of length k and send (sid, sid, sk') to P_i.
Either way, mark the record (P_i, P_j, pw) as completed.

Upon receiving (Corrupt, sid, P_i) from S: if there is a (Clt, sid, P_i, P_j, pw) recorded, return pw to S, and mark P_i corrupted. If there is a (Svr, sid, P_i, P_j, pw) recorded, then mark P corrupted (and internally) call (StealPwdFile, sid).

Functionality F_{spwKE}

The functionality F_{spwKE} 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** (StorePwdFile, sid, P_i, pw) **from party P_j:**
If this is the first StorePwdFile query, store password data record (file, P_i, P_j, pw) and mark it un compromised.

**Upon receiving a query** (CliSession, sid, sid, P_i, P_j, pw) **from party P_j:**
Send (CliSession, sid, sid, P_i, P_j, pw) to S. In addition, if this is the first CliSession query for sid, then store session record (Clt, sid, P_i, P_j, pw) and mark this record fresh.

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

### Stealing Password Data

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

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

### Active Session Attacks

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

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

### Key Generation and Authentication

**Upon receiving a query** (NewKey, sid, sid, P_i, sk) **from S, where |sk| = k:**
If there is a session record of the form (role, sid, P_i, P_j, pw) that is not marked completed,
- If this record is compromised, or either P_i or P_j is corrupted, then output (sid, sid, sk) to player P_i.
- If this record is fresh, and there is a session record (role, sid, P_j, P_i, pw') with pw' = pw, and a key sk' was sent to P_j, and (role, sid, P_j, P_i, pw) was fresh at the time, then output (sid, sid, sk') to P_i.
- In any other case, pick a new random key sk' of length k and send (sid, sid, sk') to P_i.
Either way, mark the record (P_i, P_j, pw) as completed.

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

---

### 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 Benhamouada and Pointcheval [BP13] and the novel Smooth NIZK proofs. The essential idea of [BP13] is that while the Client holds the actual
Network

Output (R)

If any of Receive (r)

Erase

Set

Choose Input (i)

Input (L)

R

Let (hkC, hpC) ← sphf(LC).hkgen and (hpS, hks) ← sphf(LS).hkgen.

Define languages:

\[ LC = \{ (R, S, H) \mid \exists r, p : R = g^r, S = a_1^i b_c^j, H = b_c^j \} \]

\[ LS = \{ (R, S) \mid \exists r : R = g^r, S = a_2^i \} \]

\[ L_C^r = \{ (R, S, H, T, l) \mid \exists r, p : R = g^r, S = a_1^i b_C^j, H = b_C^j, T = sphf.pubH(hpC, \langle R, S, H, l, r, p \rangle) \} \]

\[ L_S^r = \{ (R, S, T, l) \mid \exists r : R = g^r, S = a_2^i, T = sphf.pubH(hpS, \langle R, S, l, r \rangle) \} \]

Let \( \text{pgen}_p, \text{crs}_p, \text{prover}_p, \text{ver}_p \) be Smooth QA-NIZKs for languages \( L^r_{p} \), with \( P \in \{ C, S \} \).

Let \( \text{crs}_p \leftarrow \text{crs}_p(\rho) \) and \( H \) be a collision resistant hash function.

Let \( RO \) be a random oracle and let \( \text{phash} = RO(sid, P, P, \text{pwd}) \).

\[ \text{CRS} := \langle \rho, hp_C, hp_S, \text{crs}_S, \text{crs}_C, H \rangle. \]

Server Persistent State := \( b_s^{\text{phash}} \).

\[ R_1, S_1, T_1, H_{P_2} \rightarrow P_j \]

\[ R_2', S_2', T_2', H_{P_2}' \rightarrow P_j \]

\[ R_2, S_2, T_2, H_{P_2} \rightarrow P_1 \]

\[ R_1', S_1', T_1', H_{P_1}' \rightarrow P_i \]

**Fig. 2.** Single round RO-hybrid UC-PAKE protocol under SXDH assumption.
password, the Server does not hold password in the clear. Instead the Server
stores a hard to invert function called \( \text{PHash} \) (password hash) evaluated over the
password and a random “salt” (\( \text{PSalt} \)) published in the CRS. While executing a
session, the client sends encryptions of the password or another function called
\( \text{PPreHash} \) (password pre-hash) evaluated on the password. Correspondingly, the
server sends encryptions of the stored \( \text{PHash} \).

Of course, some kind of zero-knowledge proof must accompany these en-
cryptions, and to that end \[\text{BP13}\] can utilize the new smooth projective hash
functions (SPHF) for CCA2-encryption [BBC+13] such as Cramer-Shoup en-
cryption [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 mes-
sages use the correct password (meaning both parties have the same password
or its \( \text{PHash} \)), 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 encryp-
tion, 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 Adver-
sary can corrupt a session dynamically after the outgoing message has been sent
and the incoming message has not yet been received. Thus, this SPHF methodol-
ogy 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 gen-
erate 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 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 = bC, bS randomly sampled from G1 and a random oracle RO. Define phash = 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:

\[ \text{PPreHash}(\text{param}, \text{pwd}) = b^\text{phash}_C \]
\[ \text{PSalt}(\text{param}) = b_S \]
\[ \text{PHash}(\text{param}, \text{pwd}) = b^\text{phash}_S \]

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:

\[ L_C = \{(R, S, H) \mid \exists r, p : R = g^r, S = a^p_r b^p_C, H = b^p_S \} \]
\[ L_S = \{(R, S) \mid \exists r : R = g^r, S = a^p_r \} \]

We now plug these languages into UC-PAKE methodology described above. The client sends ElGamal encryption of b^p_C, 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 $b^p_s$, while the client divides out $b^p_s$ from the second component to form a word of $L_s$.

The CRS provides public smooth2 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:

\[
\begin{align*}
I^+_c &= \{ (R,S,H,T,l) \mid \exists r,p : R = g^r, S = a^r_i b^p_c, H = b^p_s \} \\
L^+_s &= \{ (R,S,T,l) \mid \exists r : R = g^r, S = a^r_s, T = \text{phf}.\text{pubH}(hps, (R,S,l);r) \}
\end{align*}
\]

The client generates a Smooth QA-NIZK verification key pair for the server language $L^+_s$, retains the private key $hk_1$ and sends the public key $hp_1$ along with the ElGamal encryption and the SPHF. The client computes a QA-NIZK proof $W_1$ of $(R_1, S_1, b^{\text{phash}}_s, T_1) \in L^+_s$ with label $i_1 = \mathcal{H}(\text{sid}, \text{ssid}, P_1, P_j, R_1, S_1, T_1, hp_1)$ 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 $hk_2$ and sends the public key $hp_2$ along with the ElGamal encryption and the SPHF. The server computes a QA-NIZK proof $W_2$ of $(R_2, S_2, b^{\text{phash}}_s, T_2) \in L^+_s$ with label $i_2 = \mathcal{H}(\text{ssid}, \text{sid}, P_j, P_1, R_2, S_2, T_2, hp_2)$ 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:

\[
\text{vers}.\text{privH}(hk_1, (R', S', b^{\text{phash}}_s, T', i'_1)) \cdot \text{vers}.\text{pubH}(hp', W_1).
\]

Similarly, the server computes:

\[
\text{vers}.\text{privH}(hk_2, (R', S', b^{\text{phash}}_s, T', i'_1)) \cdot \text{vers}.\text{pubH}(hp', W_2).
\]

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 $\mathcal{F}_{\text{apwKE}}$, as stated in the theorem below.

The complete protocol is described in detail in Figure 2. The SPHF $\text{phf}$ is required to be a smooth2 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 $\mathcal{F}_{\text{apwKE}}$ functionality in the $(\mathcal{F}_{\text{cns}}, \mathcal{F}_{\text{RO}})$-hybrid model, in the presence of adaptive corruption adversaries. The number of unique password
arguments passed to TestPwd and OfflineTestPwd of $F_{aprKE}$ combined in the ideal world is at most the number of random oracle calls in the $(F_{crs}, F_{RO})$-hybrid world.

6.3 Main Idea of the UC Simulator

The UC simulator $S$ works as follows: It simulates the random oracle calls and records all the query response pairs. It will generate the CRS for $F_{fake}$ 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: $S$ uses a dummy message $\mu$ 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 $S$ 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 $b_\Sigma^{\mu(c_{sid, p, pwd'}, p_1)}$, 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 $(R_1, S_1, b_\Sigma^{phash})$ in the language $L_C$, it has to use the private key of the SPHF in order to compute $T_1$ on $(R_1, S_1, b_\Sigma^{phash})$ 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 $S$ described above simulates the Adversary in the real-world protocol, follows essentially from the properties of the Smooth QA-NIZK and smooth2 SPHF, and we give a broad outline here. The proof will describe various experiments between a challenger $C$ and the adversary, which we will just assume to be the environment $Z$ (as the adversary $A$ can be assumed to be just dummy and following $Z$'s commands). In the first experiment the challenger $C$ will just be the combination of the code of the simulator $S$ above and $F_{fake}$. 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 \( S \)) 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 \( g^{r_1}, a_1^{r_1} b_c^{\mu} g^{r_1'} \) by real tuples (as in real-world) \( g^{r_1}, a_1^{r_1} b_c^{\text{phash}} \), 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 \( G_1 \).

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

Once all the instances are corrected, i.e. \( R_1, S_1 \) are generated as \( g^{r_1}, a_1^{r_1} b_c^{\text{phash}} \), the challenger can switch to the real-world because the tuples \( R_1, S_1, b_s^{\text{phash}} \) 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 \( F_{\text{PAKE}} \) leaks the password to the Adversary in the ideal world. In the protocol described above, the Adversary also gets \( W_1 \) and \( \text{HK}_1 \), as this is the only state maintained by each client between sending \( R_1, S_1, T_1, \text{HP}_1 \), and the final issuance of session-key. Simulation of \( \text{HK}_1 \) is easy for the simulator \( S \) since \( S \) generates \( \text{HK}_1 \) exactly as in the real world. For generating \( W_1 \), which \( S \) had postponed to computing till it received an incoming message from the adversary, it can now use the pwd which it gets from \( S_{\text{PAKE}} \) by issuing a Corrupt call to \( S_{\text{PAKE}} \). More precisely, it issues the Corrupt call, and gets pwd, and then calls the QA-NIZK simulator with the tuple \((R_1, S_1, b_s^{\text{phash}}, T_1, i_1)\) 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 \( \text{sk}_1 \) (which is really used in the output to the environment when \( \text{pwd}' = \text{pwd} \)).

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 \((R_2, S_2/b_s^{\text{phash}}, T_2, i_2)\).

### 6.6 Simulator for the Protocol

We will assume that the adversary \( A \) in the UC protocol is dummy, and essentially passes back and forth commands and messages from the environment \( Z \).
Thus, from now on we will use environment $Z$ as the real adversary, which outputs a single bit. The simulator $\mathcal{S}$ will be the ideal world adversary for $F_{\text{apwKE}}$. 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 $Z$ (i.e. $\mathcal{A}$). We will call a message legitimate if it was not altered by $Z$, 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 $(\text{sid}, P_1, P_2, x)$, for some $x$ and the password file has been stolen then call $\text{OfflineTestPwd}$ with $x$. If the test succeeds then return phash, which must already have been set (see Stealing Password File below), and record $(m, \text{phash})$.

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 $\tau_\mathcal{C}, \tau_\mathcal{S}$. It retains $a_1, a_2, r_\mathcal{C}, r_\mathcal{S}, h_\mathcal{K}_C, h_\mathcal{K}_S$ as trapdoors.

**New Client Session:** Sending a message to $Z$. On message ($\text{CltSession, sid, ssid, } P_1, P_2) \in F_{\text{apwKE}}$, $\mathcal{S}$ starts simulating a new instance of the protocol for client $P_1$, server $P_2$, session identifier ssid, and CRS set as above. We will denote this instance by $(P_1, \text{ssid})$ and call it a *client instance*.

To simulate this instance, $\mathcal{S}$ chooses $r_1, r_1', r_2', s_1$ at random, and sets $R_1 = g^{r_1}, S_1 = a_1^{r_1} b_\mathcal{C}^{r_1} \cdot g^{r_1'},$ and $T_1 = g^{r_1''}$ (note the use of arbitrary constant $\mu$ instead of phash). Next, $\mathcal{S}$ generates $(\text{HK}_1, \text{HP}_1) \leftarrow \text{ver.hkgen(CRS}_\mathcal{C})$ and sets $i_1 = \mathcal{H}(\text{sid, ssid, } P_1, P_2, R_1, S_1, \text{HP}_1)$. It retains $(i_1, \text{HK}_1)$. It then hands $(R_1, S_1, T_1, \text{HP}_1)$ to $Z$ on behalf of this instance.

**New Server Session:** Sending a message to $Z$. On message ($\text{SvrSession, sid, ssid, } P_1, P_2) \in F_{\text{apwKE}}$, $\mathcal{S}$ starts simulating a new instance of the protocol for client $P_1$, server $P_2$, session identifier ssid, and CRS set as above. We will denote this instance by $(P_2, \text{ssid})$ 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 $R_2 = g^{r_2}, S_2 = a_2^{r_2} b_\mathcal{S}^{r_2} \cdot g^{r_2'},$ and $T_1 = g^{r_2''}$ (note the use of arbitrary constant $\mu$ instead of phash). Next, $\mathcal{S}$ generates $(\text{HK}_2, \text{HP}_2) \leftarrow \text{ver.hkgen(CRS}_\mathcal{S})$ and sets $i_2 = \mathcal{H}(\text{sid, ssid, } P_1, P_2, R_2, S_2, \text{HP}_2)$. It retains $(i_2, \text{HK}_2)$. It then hands $(R_2, S_2, T_2, \text{HP}_2)$ to $Z$ on behalf of this instance.

**On Receiving a Message from $Z$.** On receiving a message $R'_2, S'_2, T'_2, \rho'_2$ from $Z$ intended for a client instance $(P, \text{ssid})$, the simulator $\mathcal{S}$ 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 $Z$ is same as message sent by $\mathcal{S}$ on behalf of peer $P'$ in session ssid, then $\mathcal{S}$ just issues a $\text{NewKey}$ call for $P$. 

19
3. ("Client Only Step"): If $\text{StealPwdFile}$ has already taken place then do the following: If $S'_2 = R'_{2}^{\hat{\text{apwKE}}}$, then $S$ calls $\mathcal{F}_{\text{apwKE}}$ with $(\text{Impersonate}, P, \text{ssid}, \text{pwd})$ 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 $S'_2 = R'_{2}^{\hat{\text{apwKE}}}$ and $m_x$ is of the form $(\text{sid}, P_j, \text{pwd}')$. If so, then $S$ calls $\mathcal{F}_{\text{apwKE}}$ with $(\text{TestPwd}, \text{ssid}, P, \text{pwd}')$ 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 $i'_2 = \mathcal{H}(\text{sid}, \text{ssid}, P_j, P_j, R'_2, S'_2, \mu)$.
   
   If $T'_2 \neq \text{spf}_S \mathcal{H}(hk, (R'_{2}, S'_{2}/b_{5}^{\text{phash}}), i'_2)$ then goto the step “Other Cases”. Else, compute $W_1 = \text{sim}(\text{CRS}_C, \tau_C, (R_1, S_1, b_{5}^{\text{phash}}, T_1, i_1))$. Issue a NewKey call to $\hat{\mathcal{F}}_{\text{Pake}}$ with key
   $$\text{vers}_S \mathcal{H}(\text{HK}_1, (R'_{2}, S'_{2}/b_{5}^{\text{phash}}), T'_2, i'_2)) \cdot \text{vers}_C \mathcal{H}(\text{HP}', W_1)$$

6. ("Other Cases"): $S$ issues a TestPwd call to $\hat{\mathcal{F}}_{\text{Pake}}$ with the dummy password $\mu$, 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 $R'_1, S'_1, T'_1, \text{HP}'_2$ from $Z$ intended for a server instance $(P, \text{ssid})$, the response of the simulator $S$ 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 TestPwd call by the simulator, before this 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 TestPwd call. For all the $m_k$’s of the form $(\text{sid}, P_j, \text{pwd}')$, it calls (OfflineTestPwd, sid, pwd'). Next, $S$ calls StealPwdFile. If 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 $b_{5}^{\text{phash}}$ is computed accordingly and given to the adversary.

**Client Corruption.** On receiving a Corrupt call from $Z$ for client instance $P_i$ in session $\text{ssid}$, the simulator $S$ calls the Corrupt routine of $\mathcal{F}_{\text{apwKE}}$ to obtain pwd. If $S$ had already output a message to $Z$, and not output $sk_1$ it computes

$$W_1 = \text{sim}_C(\text{CRS}_C, \tau_C, (R_1, S_1, b_{5}^{\text{phash}}, T_1, i_1)),$$

and outputs this $W_1$ along with pwd, and $\text{HK}_1$ 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 $Z$ only when pwd’ = pwd).

Without loss of generality, we can assume that in the real-world if the Adversary (or Environment $Z$) 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 $Z$) either sets the key for that session or can compute it from the internal state it broke into.
Server Corruption. On receiving a Corrupt call from \( Z \) for server instance \( P_j \) in session \( ssid \), the simulator \( S \) first performs the steps in the section on Stealing Password File above. In particular this sets the value of phash. It then calls the Corrupt routine of \( \mathcal{F}_{\text{apwKE}} \). If \( S \) had already output a message to \( Z \), and not output sk_1 it computes
\[
W_2 = \text{sim}_S(\text{CRS}_S, \tau_S, (R_2, S_2/b_{\text{phash}}^S, T_2, i_2)).
\]
and outputs this \( W_2 \) along with \( \text{hk}_2 \) 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 \( \text{OfflineTestPwd} \) is only called once for each random oracle input, which was not already tested by calling \( \text{TestPwd} \). Hence the number of unique password arguments passed to \( \text{TestPwd} \) and \( \text{OfflineTestPwd} \) of \( \mathcal{F}_{\text{apwKE}} \) 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 \( C \) and the environment \( Z \), starting with \( \text{Expt}_0 \) which we describe next. We will show that the view of \( Z \) in \( \text{Expt}_0 \) is same as its view in UC-APAKE ideal-world setting with \( Z \) interacting with \( \mathcal{F}_{\text{apwKE}} \) and the UC-PAKE simulator \( S \) 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 \( \mathcal{F}_{\text{apwKE}} \) by the protocol \( \Pi \).

Here is the complete code in \( \text{Expt}_0 \) (stated as it’s overall experiment with \( Z \)):

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 \( r \leftarrow Z_q \), record \((m, r)\) and return \( r \).

2. The challenger \( C \) 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 \( \tau_C, \tau_S \). It retains \( a_1, a_2, \tau_C, \tau_S, \text{hk}_C, \text{hk}_S \) as trapdoors.

Next, (on \( \text{StorePwdFile} \)) the challenger calls the random oracle with query \((sid, P_i, P_j, pwd)\). It sets phash equal to the random oracle response and sets the server persistent state as \( b_{\text{phash}}^S \).

Define \( \text{PhashIsSet} \) to be true after either \( \text{StealPwdFile} \) has been called or the random oracle has been called with \((sid, P_i, P_j, pwd)\) by the adversary, and false before.

Define \( \text{PwdCalled} \) to be true after the random oracle has been called with \((sid, P_i, P_j, pwd)\) by the adversary, and false before.
3. On receiving (CltSession, sid, ssid, P1, P2) from Z, C generates (HK1, HP1) ←
\text{vers}_{hkgen}(crsS). Next, C chooses r1, r1′, r1″, s1 at random, and sets 
R1 = g^{r1}, S1 = a1^R b1^C g^{r1′} and T1 = g^{r1″}. It then hands (R1, S1, T1, HP1) to Z on behalf 
of this instance.

4. On receiving (R2′, S2′, T2′, HP2′) from Z, intended for client session (P1, 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 sk1 ← GT.
(b) If the message received is identical to message sent by C in the same 
session (i.e. same ssid) on behalf of the peer, then output sk1 ← GT
(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 
sk1 here).
(c) If PHASHISSET is false, then output sk1 ← GT.
(d) If S2′ ≠ R2′a2b2^phash, then output sk1 ← GT.
(e) At this point we must have S2′ = R2′a2b2^phash.
Compute: i′ 2 = \mathcal{H}(sid, ssid, P1, P1, R2′, S2′, HP2′).
If T2′ ≠ sphfS.privH(hk1, (R2′, S2′/b2^phash), i′ 2) then output sk1 ← GT.
Else, compute W1 = \text{sim}_{C}(crsC, \tau_C, (R1, S1, bS^phash, T1, i1)). Output: 
\text{sk1} = \text{vers}_{C}.privH(HK1, (R2′, S2′/b2^phash, T2′, i′ 2)) \cdot \text{ver}_{C}.pubH(HP2′, W1)

5. On a Corrupt call for client P1, output pwd. If Step 3 has already happened
then also output HK1 and W1 = \text{sim}_{C}(crsC, \tau_C, (R1, S1, bS^phash, T1, i1)).

6. On receiving (SrvSession, sid, ssid, Pj, P1) from Z, follow steps symmetric 
to Step 4, swapping subscripts and languages accordingly and replacing the 
condition PHASHISSET by PWD CALLED in Step 4c.

7. On a Corrupt call for server Pj, if Step 3 has already happened then output 
HK2, and W2 = \text{sim}_{S}(crsS, \tau_S, (R2, S2/bS^phash, T2, i2)). Finally, execute a 
StealPwdFile call, as described below.

8. On a StealPwdFile call, return bS^phash as the Server Persistent State to the 
adversary.

All outputs of sk1 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 C 
outputs R1, S1, ... to Z, and a phase where it receives a message from Z. However, C
cannot output sk1 until it has completed both phases. These orderings
are dictated by Z. We will consider two different kinds of temporal orderings. A
temporal ordering of different instances based on the order in which C outputs
sk1 in an instance will be called \textbf{temporal ordering by key output}. A
temporal ordering of different instances based on the order in which C outputs its
first message (i.e. R1, S1, ... ) will be called \textbf{temporal ordering by message output}. It is easy to see that C can dynamically compute both these orderings
by maintaining a counter (for each ordering).
We now claim that the view of \( Z \) in Expt_0 is statistically indistinguishable from its view in its combined interaction with \( F_{\text{apKE}} \) and \( S \). The CRS is set identically by both \( C \) and \( S \). While \( C \) has access to pwd from the outset and sets up the random oracle output phash corresponding to \((\text{sid}, P_i, P_j, \text{ssid})\) at the beginning, \( S \) 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 \( S \) 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 \( C \) and \( S \) respond to random oracle queries identically.

Both \( C \) and \( S \) generate the client and server flows identically. In particular, observe that the condition \text{PhashIsSet} exactly captures the state of \( S \) for a client session where it knows phash and can compute the relevant elements and keys. \( C \) uses the condition \text{PhashIsSet} to do the same computations. Similarly for the server sessions with the condition \text{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 \( C \) and \( S \) end up computing keys identically. While \( C \) directly checks by exponentiation with phash in the case that pwd was guessed correctly, \( S \) 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 \text{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 \( b_s^{\text{phash}} \), there is negligible chance of it passing the condition \( S'_1 = R'^{a_1}_1 b'^{\text{phash}}_s \), unless it queries the random oracle with the correct password. This can be proved by employing DDH on \((b_s, b_C, b_s^{\text{phash}}, b_C^{\text{phash}})\). 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 \((b_s, b_C, b_s^{\text{phash}}, b_C^{\text{phash}})\). In particular the condition \( S'_1 = R'^{a_1}_1 b'^{\text{phash}}_s \) can be switched by DDH to \( S'_1 = R'^{a_1}_1 b'^{\text{phash}}_C \), 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 \( T'_2 \neq \operatorname{sphfs}_s \cdot \operatorname{privH}(hks, (R'_2, S'_2/b_s^{\text{phash}}, i'_2)) \) in Step 4e in
Expt\(_0\) is replaced by:

\[(S'_2 \neq R'_a b^\text{phash}_s) \text{ or } (T'_2 \neq \text{sphf}_s.\text{privH}(hks_s, (R'_2, S'_2/b^\text{phash}_s), i'_2)).\]

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 \(T'_2 \neq \text{sphfs}.\text{privH}(hks, (R'_2, S'_2/b^\text{phash}_s), i'_2),\) i.e. the disjunct \((S'_2 \neq R'_a b^\text{phash}_s)\) is dropped.

First note that \(T'_1\) is being computed randomly. The experiment \(\text{Expt}_3\) is then statistically indistinguishable from \(\text{Expt}_2\) by smoothness of \(\text{sphfs}\) (note that it can be shown that the polynomial number of extra bits of information leaked by the conditions \(T'_2 \neq \text{sphfs}.\text{privH}(hks, (R'_2, S'_2/b^\text{phash}_s), i'_2)\) themselves have a negligible effect on the smoothness of \(\text{sphfs}\) – 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 \(a^i r^i_1 b^\text{phash}_s \cdot g^i_1\). Symmetrically, for the server instance. Note the use of phash instead of \(\mu\).

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: \(a^i r^i_1 b^\text{phash}_s\). Further, \(T_1\) is computed as follows: \(T_1 = \text{sphf}_c.\text{pubH}(\text{hp}_c, (R_1, S_1, b^\text{phash}_s), i_1; r_1, \text{phash}).\) Symmetrically, for the server instances.

To show that \(\text{Expt}_4\) is computationally indistinguishable from \(\text{Expt}_5\), we define several hybrid experiments \(\text{Expt}_{4,i}\) inductively. Experiment \(\text{Expt}_{4,0}\) is identical to \(\text{Expt}_4\). If there are a total of \(N\) instances, \(\text{Expt}_{4,N}\) will be identical to \(\text{Expt}_5\). Experiment \(\text{Expt}_{4,i+1}\) differs from experiment \(\text{Expt}_{4,i}\) in only (temporally ordered by message output) the \((i+1)\)-th instance. While in \(\text{Expt}_{4,i}\), the \((i+1)\)-th instance is simulated by \(C\) as in \(\text{Expt}_4\), in \(\text{Expt}_{4,i+1}\) this instance is simulated as in \(\text{Expt}_5\).

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

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

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

\[T_1 = \text{sphf}_c.\text{privH}(\text{hp}_c, (R_1, S_1, b^\text{phash}_s), i_1)\]  \hspace{1cm} (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 \(\text{hp}\)
(in instances less than \((i + 1)\)). Then the claim follows by smoothness of \(\text{sphf}_c\), and noting that \(S_1 \neq R_i^{b_i} b_i^{\text{phash}}\) in instance \((i + 1)\) (by construction of \(\text{Expt}_{4,i}\)).

In the next experiment \(G_3\), the challenger generates the \(S_1\) in the \((i + 1)\)-th instance as follows: \(S_1 = a_i^{r_i} b_i^{\text{phash}}\). That the view of \(Z\) in experiments \(G_1\) and \(G_2\) are computationally indistinguishable follows from the DDH assumption in group \(G_1\) (note \(a_1\) is not being used by the challenger, now that Step 4d is no more).

In the next experiment \(G_3\), change the computation of \(T_1\) in session \((i + 1)\) to use the public hash (of \(\text{sphf}_c\)) and witness \(r_1\). Since, now \((R_1, S_1, b_2^{\text{phash}})\) is in the language \(L_c\), indistinguishability from the previous experiment follows by correctness of \(\text{sphf}_c\).

\(\text{Expt}_6\) : In this experiment, the crs is generated using \(\text{crsgen}\) instead of the crs-simulator, and \(W_1\) is computed everywhere by \(\text{prover}\) 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 members. At this point, the complete experiment \(\text{Expt}_6\) 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 \(C\) picks the CRS just as in the real world. It retains \(a_1, a_2, h_k_c\), \(h_k_s\) as trapdoors. Next the challenger calls the random oracle with query \((\text{sid}, P, P_j, \text{pwd})\). It sets \(\text{phash}\) equal to the random oracle response and sets the server persistent state as \(b_2^{\text{phash}}\).
3. On receiving \((\text{CltSession}, \text{sid}, \text{ssid}, P_i, P_j)\) from \(Z\), \(C\) generates \((h_k_1, h_p_1) \leftarrow \text{vers.hkgen(crs}_Z\)). Next, \(C\) chooses \(r_1\) at random, and sets \(R_1 = g^{r_1}\), \(S_1 = a_i^{r_1} b_i^{\text{phash}}\) and \(T_1 = \text{sphf}_C\cdot \text{pubH}(h_p_c, (R_1, S_1, b_i^{\text{phash}}), i_1, r_1, \text{phash})\). It then hands \((R_1, S_1, T_1, h_p_1)\) to \(Z\) on behalf of this instance.
4. On receiving \((R_2', S_2', T_2', h_p_2')\) from \(Z\), intended for client session \((P_i, \text{ssid})\) (and assuming no corruption of this instance):
   
   a) If the received elements are either not in their respective groups, or are trivially 1, output \(s_k_1 \leftarrow \mathbb{G}_T\).
   
   b) If the message received is identical to message sent by \(C\) in the same session (i.e. same \(\text{ssid}\)) on behalf of the peer, then output \(s_k_1 \leftarrow \mathbb{G}_T\) (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 \(s_k_1\) here).
   
   c) \(\text{Compute: } i'_2 = \mathcal{H}(\text{sid, ssid, P}_j, P_i, R_2', S_2', h_p_2').\)

   If \(T_2' \neq \text{sphf}_s\cdot \text{privH}(h_k_s, (R_2', S_2'/b_2^{\text{phash}}, i'_2))\) then output \(s_k_1 \leftarrow \mathbb{G}_T\).

   Else, compute \(W_1 = \text{prover}_{\text{crs}_C, (R_1, S_1, b_2^{\text{phash}}, T_1, i_1)})\). Output:

   \[
   s_k_1 = \text{vers}_{\text{crs}_C, (R_2', S_2'/b_2^{\text{phash}}, T_2', i_2')} \cdot \text{vers}_{\text{pubH}(h_p_2', W_1)}
   \]
5. On a **Corrupt** call for client $P_i$, output pwd. If Step 3 has already happened then also output $h_k_1$ and $W_1 = prover_c(crs_c, \langle R_1, S_1, b_s^{\text{hash}}, T_1, i_1 \rangle)$.

6. On receiving $(\text{SrvSession}, \text{sid}, \text{ssid}, P_j, P_i)$ from $Z$, follow steps symmetric to Step 4, swapping subscripts and languages accordingly.

7. On a **Corrupt** call for server $P_j$, if Step 3 has already happened then output $h_k_2$, and $W_2 = provers(srs_s, \langle R_2, S_2/b_s^{\text{hash}}, T_2, i_2 \rangle)$. Finally, execute a StealPwdFile call, as described below.

8. On a **StealPwdFile** call, return $b_s^{\text{hash}}$ 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 $C$ 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 **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 **Expt_{6,0}** is same as experiment **Expt_6**, and experiment **Expt_{6,N}** is same as experiment **Expt_7**.

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

*Proof.* The lemma is proved using several hybrid experiments of its own. The experiment **H_0** is same as **Expt_{6,i-1}**.

In experiment **H_1**, the CRS is set as in the real world, except that the QA-NIZK $\text{crs}_c$ is set using the crs simulators $\text{crssim}_c$ (the challenger retains the trapdoors $\tau_c$ output by the crs simulator). All proofs $W_1$ are still computed using $\text{prover}_c$. Experiments **H_0** and **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 **H_2**, in instance $i$, the value $W_1$ (in Step 4e or corruption) is generated using the proof simulator using trapdoor $\tau$. Indistinguishability follows by zero-knowledge property of the QA-NIZK as the proof being generated is on a language member.

In experiment **H_3**, in instance $i$, the value $T_1$ is generated using the private hash key $h_k_c$, and the private hash function $\text{sphf}_c, \text{privH}$ (thus eliminating the use of witness $r_1$). Experiments **H_2** and **H_3** are indistinguishable by the correctness of $\text{sphf}_c$.

In experiment **H_4**, in instance $i$, the values $R_1, S_1$ are generated as $R_1 = g^{a_1}, S_1 = a_1^{r_1}b_c^{\text{hash}} \cdot g^{a_1},$ where $r_1, r_1'$ are random and independent. This follows by employing DDH on $g, g^{a_1}, a_1$ and either $g^{a_1r_1}$ or $g^{a_1r_1+r_1'}$. 

26
In experiment $H_5$, in peer of instance $i$, in Step 4e the condition $T'_1 \neq \text{sphfc}_C.\text{privH}(hk_C, (R'_1, S'_1, b_C^{\text{hash}}, i'_1))$ is replaced by $(S'_1 \neq R'_1^{\alpha}, b_C^{\text{hash}})$ or $T'_1 \neq \text{sphfc}_C.\text{privH}(hk_C, (R'_1, S'_1, b_C^{\text{hash}}, i'_1))$. Indistinguishability from experiment $H_4$ follows by smooth$_2$ property of sphfc$_C$, noting that at most one bad sphfc$_C.\text{privH}$ is being output to the Adversary (namely $T_1$ in instance $i$).

In experiment $H_6$, in instance $i$, change Step 4b as follows: If the message received is identical to message sent by $C$ 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 $i'_2 = \mathcal{H}(sid, ssid, P, P_j, T, i'_1)$, Output

$$\text{vers} \cdot \text{privH}(hk_2, (R'_2, S'_2/b_2^{\text{hash}}, T'_2, i'_2)) \cdot \text{verc} \cdot \text{privH}(hk_2, (R'_1, S_1, b_2^{\text{hash}}, T_1), i_1).$$

Here $hk_2$ is the HK output by vers.hkgen in the peer instance of instance $i$.

The experiments $H_6$ and $H_5$ are computationally indistinguishable by noting the following three facts:

1. In the peer of instance of instance $i$ (which generated $hk_2$), in Step 4e the computation $\text{verc} \cdot \text{privH}(hk_2, \cdot)$ 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 crsgen$_C$.
3. Then, $\text{verc} \cdot \text{privH}(hk_2, (R'_1, S_1, b_2^{\text{hash}}, T_1), i_1)$ is random even when the adversary is given HP$_2$ by smoothness of the QA-NIZK, since $S_1 \neq R_1^{\alpha}, b_2^{\text{hash}}$.

In experiment $H_7$, in peer of instance $i$, in Step 4e the condition “$(S'_1 \neq R'_1^{\alpha}, b_C^{\text{hash}})$ or $T'_1 \neq \text{sphfc}_C.\text{privH}(hk_C, (R'_1, S'_1, b_C^{\text{hash}}, i'_1))$” is replaced by “$T'_1 \neq \text{sphfc}_C.\text{privH}(hk_C, (R'_1, S'_1, b_C^{\text{hash}}, i'_1))$”. Indistinguishability from experiment $H_6$ follows by smooth$_2$ property of the sphfc$_C$, noting that at most one bad sphfc$_C.\text{privH}$ is being output to the Adversary (namely $T_1$ in instance $i$).

In experiment $H_8$, in instance $i$, $R_1, S_1$ are generated as $R_1 = g^{x_1}, S_1 = a_1^{\ast}b_C^{\text{hash}}$, by employing DDH.

In experiment $H_9$, 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 $H_{10}$, 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 $H_{11}$, in Step 4b the key is output as follows:

- Else, compute $i'_2 = \mathcal{H}(sid, ssid, P, P_j, R'_2, S'_2, HP'_2)$.

Compute $W_1 = \text{proverc}(\text{crsc}_C, (R'_1, S_1, b_2^{\text{hash}}, T_1, i_1), r_1)$. Output

$$\text{vers} \cdot \text{privH}(hk_1, (R'_2, S'_2/b_2^{\text{hash}}, T'_2, i'_2)) \cdot \text{verc} \cdot \text{pubH}(HP'_2, W_1)$$
Indistinguishability follows by noting that $hP'_2$ is exactly the $hP_2$ 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

Expt$_8$: In this experiment in Step 4e the condition is changed to “$(S'_2 \neq R''_2b^\text{phash})$ or $T'_2 \neq \text{sphf}_s\text{privH}(hk_s, (R''_2, S'_2/b^\text{phash}, i'_2))$”. In other words, the disjunct $(S'_2 \neq R''_2b^\text{phash})$ is introduced.

Indistinguishability follows by the same argument as employed in experiments Expt$_3$ and Expt$_2$.

Expt$_9$: In this experiment Step 4e is dropped altogether.

We first show that if the condition:

$$(S'_2 \neq R''_2b^\text{phash}) \text{ or } T'_2 \neq \text{sphf}_s\text{privH}(hk_s, (R''_2, S'_2/b^\text{phash}, i'_2))$$

holds, then $(R''_2, S'_2/b^\text{phash}, T'_2, i'_2)$ 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 $(S'_2 = R''_2b^\text{phash})$, with witness $r_2$ for $R''_2$. Then, by correctness of the sphf,

$$\text{sphf}_s\text{privH}(hk_s, (R''_2, S'_2/b^\text{phash}, i'_2)) = \text{sphf}_s\text{pubH}(hp_s, (R''_2, S'_2/b^\text{phash}, i'_2; r_2)).$$

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

Thus, $\text{vers}_s\text{privH}(hk_1, (R''_2, S'_2/b^\text{phash}, T'_2, i'_2))$ is random, even when the Adversary is given $hp_1$, by smooth-soundness of the QA-NIZK.

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 $Z$, since if both peers (of a instance) received legitimate messages forwarded by $Z$, then Step 4e computes the same instance key in both instances.

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

References


A Proof of Theorem 1

**Theorem 1.** (re-stated) If a matrix distribution $D_k$ on $\mathbb{Z}_q^{(k+1) \times k}$ is boostable to a matrix distribution $D_{l,k}$ on $\mathbb{Z}_q^{l \times k}$ then the $D_k$-MDDH assumption implies the $D_{l,k}$-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 $[B_s]$ to uniformly random. We start off with $i = 0$, where we have the real output $[B_s]$ 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 $([B], [b])$ is computed as follows. We sample $[A]$ from $D_k$ and $s$ from $\mathbb{Z}_q^k$. We set $[B]$ as $[A]$ and, if $i \neq 0$, the row $i$ of $[B]$ as the row $i$ of $F[A]$. All other rows $j \neq i$ of $[B]$ are set to the $j$-th row of $E[\bar{A}]$. We set the top $k$ elements of $[b]$ to be $[\bar{A}^s]$ and choose all the $(k+j)$-th elements, where $j < i$, of $[b]$ uniformly at random from $\mathbb{Z}_q$. If $i \neq 0$, we set the $(k+i)$-th element of $[b]$ to be the $i$-th element of $F[As]$. For all $j > i$, we set the $(k+j)$-th element of $[b]$ to be the $j$-th element of $E[\bar{A}s]$. To summarize, $[b]$ is computed as:

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

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

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

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

As shown above, the hybrid $i'$ is indistinguishable from hybrid $i$ by the $D_k$-MDDH assumption. Next we transition to the hybrid $i+1$ where $[b]$ is computed.
The hybrid $i + 1$ is indistinguishable from hybrid $i'$, as $E\bar{A}$ is identically distributed as $FA$. The theorem is thus established by chaining all the hybrids.

B Hardness Assumptions

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

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

Note that this is a $D_1$-MDDH assumption with the matrix $A$ being the $2 \times 1$ matrix which is the transpose of $(a \ 1)$.

Definition 4 (SXDH [BBS04]). Consider a generation algorithm $G$ taking the security parameter as input, that outputs a tuple $(q, G_1, G_2, G_T, e, g_1, g_2)$, where $G_1, G_2$ and $G_T$ are groups of prime order $q$ with generators $g_1, g_2$ and $e(g_1, g_2)$ respectively and which allow an efficiently computable $\mathbb{Z}_q$-bilinear pairing map $e : G_1 \times G_2 \rightarrow G_T$. The Symmetric eXternal decisional Diffie-Hellman (SXDH) assumption asserts that the Decisional Diffie-Hellman (DDH) problem is hard in both the groups $G_1$ and $G_2$.

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 $F_{\text{pake}}$

The functionality $F_{\text{pake}}$ 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** (NewSession, sid, $P_i$, $P_j$, pw, role) **from party** $P_i$:
Send (NewSession, sid, $P_i$, $P_j$, role) to $S$. In addition, if this is the first NewSession query, or if this is the second 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** (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 compromised and reply to $S$ with “correct guess”. If $pw \neq pw'$, mark the record interrupted and reply with “wrong guess”.

**Upon receiving a query** (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 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** (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 $F_{\text{pake}}$.

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 interrupted and the parties are provided random and independent session keys. If however the adversary makes a successful guess, then the session is marked 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 $F_{\text{pake}}$ 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 $F$ also implicitly defines the multi-session extension of $F$ (denoted by $\hat{F}$): $\hat{F}$ runs multiple independent copies of $F$, where the copies are distinguished via sub-session IDs ssid. The JUC theorem [CR03] asserts that for any protocol $\pi$ that uses multiple independent copies of $F$, composing $\pi$ instead with a single copy of a protocol that realizes $\hat{F}$, preserves the security of $\pi$. 

33
Generate $g_1 \leftarrow G_1$, $g_2 \leftarrow G_2$ and $a = g_1^a$ with $a \leftarrow Z_q$ as DH parameters $\rho$.
Let $H$ be a CRHF, and $\text{sphf}$ be a smooth$_2$ SPHF family for the DH family.

\[(hp, hk) \leftarrow \text{sphf}.hkgen(\rho).\]

Let $(\text{pargen, crsgen, prover, ver})$ be a Smooth QA-NIZK for language $L$,
\[L = \{R, S, T, l : \exists r, R = g_1^r, S = a^r, T = \text{sphf}.pubH(hp, (R, S), l, r)\}.
\]

\[\text{CRS} := (\rho, hp, crs, H).\]

<table>
<thead>
<tr>
<th>Party $P_i$</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input $(\text{NewSession, sid, ssid, P}_1, P_2, pwd, \text{initiator/responder})$</td>
<td>[\begin{array}{l} R_1, S_i, T_1, \text{HP}_1 \rightarrow P_j \end{array}]</td>
</tr>
<tr>
<td>Choose $r_1 \leftarrow Z_q$, $(\text{HK}_1, \text{HP}_1) \leftarrow \text{ver}.hkgen(\text{CRS})$.</td>
<td></td>
</tr>
<tr>
<td>Set $R_1 = g_1^{r_1}$, $S_1 =pwd \cdot a^{r_1}$, $T_1 = \text{sphf}.pubH(hp, (R_1, S_1, pwd), i_1; r_1)$, $W_1 = \text{prover}(\text{crs}, (R_1, S_2, T_1, i_1); r_1)$, where $i_1 = H(sid, ssid, P_1, P_2, R_1, S_1, \text{HP}_1)$.</td>
<td></td>
</tr>
<tr>
<td>Erase $r_1$, send $(R_1, S_1, T_1, \text{HP}_1)$ and retain $(W_1, \text{HK}_1)$</td>
<td></td>
</tr>
<tr>
<td>Receive $R'_2, S'_2, T'_2, \text{HP}'_2$.</td>
<td>[\begin{array}{l} R'_2, S'_2, T'_2, \text{HP}'_2 \rightarrow P_j \end{array}]</td>
</tr>
<tr>
<td>If any of $R'_2, S'_2, T'_2, \text{HP}'_2$ is not in their respective group or is 1, set $sk_1 \leftarrow G_T$, else</td>
<td></td>
</tr>
<tr>
<td>compute $i'_2 = H(sid, ssid, P_1, P_2, R'_2, S'_2, \text{HP}'_2)$</td>
<td></td>
</tr>
<tr>
<td>Compute $sk_1 = \text{ver}.priv(H(\text{HK}_1, (R'_2, S'_2, pwd, T'_2, i'_2)) \cdot \text{ver}.pubH(\text{HP}'_2, W_1)$.</td>
<td></td>
</tr>
<tr>
<td>Output $(sid, ssid, sk_1)$</td>
<td></td>
</tr>
</tbody>
</table>

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 $\tilde{F}_{\text{PAKE}}$.

**Theorem 4.** Assuming the existence of SXDH-hard groups, the protocol in Fig 4 securely realizes the $\tilde{F}_{\text{PAKE}}$ functionality in the $F_{\text{CRS}}$ 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 $Z$. Thus, from now on we will use environment $Z$ as the real adversary, which outputs a single bit. The simulator $S$ will be the ideal world adversary for $\tilde{F}_{\text{PAKE}}$. 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 $Z$ (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 $Z$, and delivered in the correct session and to the correct party.
The simulator $S$ picks the CRS just as in the real world, except the QA-NIZK CRS is generated using the crs-simulator, which also generates a simulator trapdoor $\tau$. It retains $a, \tau, \text{hk}$ as trapdoors.

The next main difference in the simulation of the real world parties is that $S$ uses a dummy message $\mu$ 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 $\text{pwd}'$, 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 $Z$.** On message $(\text{NewSession}, \text{sid}, \text{ssid}, i, j, \text{role})$ from $\hat{\Phi}_{\text{PAKE}}$, $S$ starts simulating a new instance of the protocol $\Pi$ for party $P_i$, peer $P_j$, session identifier $\text{ssid}$, and $\text{CRS}$ set as above. We will denote this instance by $(P_i, \text{ssid})$. To simulate this instance, $S$ chooses $r_1, r'_1, r''_1$ at random. Also, $(\text{HK}_1, \text{HP}_1) \leftarrow \text{ver.hkgen}(\text{CRS})$. It sets $R_1 = g_1^{r_1}$, $S_1 = \mu \cdot a^{r_1} \cdot g_1^{r'_1}$, $T_1 = g_1^{r''_1}$. Let $t_1 = H(\text{sid}, \text{ssid}, i, j, R_1, S_1, \text{HP}_1)$. (Note the use of $\mu$ instead of $\text{pwd}$).

It retains $r_1, r'_1, r''_1, t_1, \text{HK}_1$ (and $\mu$ if chosen randomly). It then hands $R_1, S_1, T_1, \text{HP}_1$ to $Z$ on behalf of this instance.

**On Receiving a Message from $Z$.** On receiving a message $R'_2, S'_2, T'_2, \text{HP}'_2$ from $Z$ intended for this instance $(P_i, \text{ssid})$, the simulator $S$ makes the real world protocol checks, namely group membership and non-triviality. If any of these checks fail, it issues a $\text{TestPwd}$ call to $\hat{\Phi}_{\text{PAKE}}$ with the dummy password $\mu$, followed by a $\text{NewKey}$ 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 $Z$ is same as message sent by $S$ on behalf of peer $P_j$ in session $\text{ssid}$, then $S$ just issues a $\text{NewKey}$ call for $P_i$.

Else, it computes $\text{pwd}'$ by decrypting $S'_2$, i.e. setting it to $S'_2/(R'_2)^a$. $S$ then calls $\hat{\Phi}_{\text{PAKE}}$ with $(\text{TestPwd}, \text{ssid}, P_i, \text{pwd}')$. Regardless of the reply from $F$, it then issues a $\text{NewKey}$ call for $P_i$ with key computed as follows (recall, $R_1, S_1, t_1, r'_1, r''_1$ from earlier in this instance when the message was sent to $Z$). Let,

$$\ell'_2 = H(\text{sid}, \text{ssid}, P_j, P_i, R'_2, S'_2, \text{HP}'_2),$$

$$W_1 = \text{sim} \left( \text{CRS}, \tau, (R_1, S_1/\text{pwd}', T_1, t_1) \right)$$

If $T'_2 \neq \text{roph}.\text{privH}(\text{HK}, (R'_2, (R'_2)^a), \ell'_2)$ then call $\hat{\Phi}_{\text{PAKE}}$’s $\text{NewKey}$ with a random key else call $\text{NewKey}$ with key

$$\text{ver}.\text{privH}(\text{HK}_1, (R'_2, (R'_2)^a, T'_2, \ell'_2)) \cdot \text{ver}.\text{pubH}(\text{HP}'_2, W_1).$$

By definition of $\hat{\Phi}_{\text{PAKE}}$, this has the effect that if the $\text{pwd}'$ was same as the actual $\text{pwd}$ previously recorded in $\hat{\Phi}_{\text{PAKE}}$ (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.

35
Corruption  On receiving a Corrupt call from $Z$ for instance $P_i$ in session ssid, the simulator $S$ calls the Corrupt routine of $\hat{F}_{PAKE}$ to obtain pwd. If $S$ had already output a message to $Z$, and not output sk$_1$ (via a call to NewKey) it computes

$$W_1 = \text{sim}(\text{crs}, \tau, (R_1, S_1/pwd, T_1, t_1)),$$

and outputs this $W_1$ along with pwd, and 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 $Z$ only when pwd$' = pwd$).

Without loss of generality, we can assume that in the real-world if the Adversary (or Environment $Z$) 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 $Z$) 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 $C$ and the environment $Z$, starting with Expt$_0$ which we describe next. We will show that the view of $Z$ in Expt$_0$ is same as its view in UC-PAKE ideal-world setting with $Z$ interacting with $\hat{F}_{PAKE}$ and the UC-PAKE simulator $S$ 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 $F_{PAKE}$ by the protocol $\Pi$.

Here is the complete code in Expt$_0$ (stated as it’s overall experiment with $Z$):

1. The challenger $C$ picks the CRS just as in the real world, except the QA-NIZK crs is generated using the crs-simulator crssim, which also generates a simulator trapdoor $\tau$. $C$ retains $a, \tau, hk$.

2. On receiving NewSession, sid, ssid, $P_i, P_j$, pwd, role from $Z$, $C$ generates $(hk_1, hp_1)$ by running ver.hkgen(crs). Next, it generates $R_1, S_1, T_1$ by choosing $r_1, r'_1, r''_1$ at random, and setting $R_1 = g^{r_1}, S_1 = a^{r'1} \cdot g^{r''1}, T_1 = g^{r''1}$. It sends these values along with $hp_1$ to $Z$.

3. On receiving $R'_2, S'_2, T'_2, hp_2$ from $Z$, 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 $G_T$.
   
   (b) Otherwise, if the message received is identical to message sent by $C$ in the same session (i.e. same ssid) on behalf of the peer, then output sk$_1 \leftarrow G_T$ (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 pwd$' = S'_2/(R'_2)^a$. If pwd$' \neq pwd$ (note pwd was given in NewSession request), then output sk$_1$ randomly and independently from $G_T$. 

36
(d) Else, compute \( \ell_2' = \mathcal{H}(\text{sid}, \text{ssid}, P_j, P_i, R_2', S_2', \text{hp}'_2) \).

If \( T'_2 \neq \text{spfh. privH}(\text{hk}, (R'_2, S'_2/\text{pwd}), \ell'_2) \) then output a random value in \( \mathbb{G}_T \).

Else, compute \( W_1 = \text{sim}(\text{crs}, \tau, (R_1, S_1/\text{pwd}, T_1, \iota_1)), \) where \( \iota_1 = \mathcal{H}(\text{sid}, \text{ssid}, P_i, P_j, R_1, S_1, \text{hp}_1) \), and output

\[
\text{ver. privH}(\text{hk}_1, (R'_2, S'_2/\text{pwd}, T_2, \ell'_2)) \cdot \text{ver. pubH}(\text{hp}'_2, W_1).
\]

4. On a Corrupt call, if Step 2 has already happened then output \( \text{HK}_1, \text{pwd} \) and \( W_1 = \text{sim}(\text{crs}, \tau, (R_1, S_1/\text{pwd}, T_1, \iota_1)) \).

All outputs of \( \text{sk}_1 \) are also accompanied with \( \text{sid}, \text{ssid} \) (but are not mentioned above for ease of exposition).

Note that each instance has two asynchronous phases: a phase in which \( \mathcal{C} \) outputs \( R_1, S_1, \ldots \) to \( \mathcal{Z} \), and a phase where it receives a message from \( \mathcal{Z} \). However, \( \mathcal{C} \) cannot output \( \text{sk}_1 \) 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 \( \text{sk}_1 \) 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. \( R_1, S_1, \ldots \)) 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 \( \text{Expt}_0 \) is identical to its view in its combined interaction with \( \tilde{\mathcal{F}}_{\text{PKE}} \) and \( \mathcal{S} \), as \( \mathcal{C} \) has just combined the code of \( \tilde{\mathcal{F}}_{\text{PKE}} \) and \( \mathcal{S} \) (noting that in Step 3(d), \( \text{pwd} = \text{pwd}' \))

\[
\text{Expt}_1 : \text{In this experiment Step 3(c) is dropped altogether and Step 3(d) altered as follows: In Step 3(d) in Expt}_0, \text{the condition } T'_2 \neq \text{spfh. privH}(\text{hk}, (R'_2, S'_2/\text{pwd}), \ell'_2) \text{ is replaced by } \text{"if } (S'_2 \neq \text{pwd} \cdot (R'_2)^a) \text{ or } T'_2 \neq \text{spfh. privH}(\text{hk}, (R'_2, S'_2/\text{pwd}), \ell'_2) \text{".}
\]

Rest of the computation of \( \text{sk}_1 \) in Step 3(d) remains the same.

We claim that the view of \( \mathcal{Z} \) is statistically identical in \( \text{Expt}_0 \) and \( \text{Expt}_1 \). This follows by noting that \( S'_2 \neq \text{pwd} \cdot (R'_2)^a \) is equivalent to the condition \( \text{pwd}' \neq \text{pwd} \) in \( \text{Expt}_0 \). The condition \( S'_2 = \text{pwd} \cdot (R'_2)^a \) held in Step 3(d) in \( \text{Expt}_0 \), as that step was only reached if this condition held.

\[
\text{Expt}_2 : \text{In this experiment, in Step 3(d) the condition is replaced by just } \text{"if } T'_2 \neq \text{spfh. privH}(\text{hk}, (R'_2, S'_2/\text{pwd}), \ell'_2) \text{", i.e. the disjunct } (S'_2 \neq \text{pwd} \cdot (R'_2)^a) \text{ is dropped.}
\]

First note that \( T_1 \) is being computed randomly. The experiment \( \text{Expt}_2 \) is then statistically indistinguishable from \( \text{Expt}_1 \) by smoothness of \( \text{spfh} \) (note that it can be shown that the polynomial number of extra bits of information leaked by the conditions \( T'_2 \neq \text{spfh. privH}(\text{hk}, (R'_2, S'_2/\text{pwd}), \ell'_2) \) themselves have a negligible effect on the smoothness of the \( \text{spfh} \) – 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 $pwd \cdot a^{r_1} \cdot g_1^{r_1'}$. Note the use of $pwd$ instead of $\mu$.

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: $pwd \cdot a^{r_1}$. Further, $T_1$ is computed as follows: $T_1 = \text{sphf}.\text{pubH}(hp, \langle R_1, S_1/pwd \rangle, \iota_1)$.

To show that **Expt**$_4$ is computationally indistinguishable from **Expt**$_3$, we define several hybrid experiments **Expt**$_3$,inductively. Experiment **Expt**$_3,0$ is identical to **Expt**$_3$. If there are a total of $N$ instances, **Expt**$_3,N$ will be identical to **Expt**$_4$. Experiment **Expt**$_3,i+1$ differs from experiment **Expt**$_3,i$ in only (temporally ordered by message output) the $(i+1)$-th instance. While in **Expt**$_3,i$, the $(i+1)$-th instance is simulated by $C$ as in **Expt**$_3$, in **Expt**$_3,i+1$ this instance is simulated as in **Expt**$_4$.

**Lemma 3.** For all $i : 0 \leq i \leq N$, the view of $Z$ in experiment **Expt**$_3,i+1$ is computationally indistinguishable from the view of $Z$ in **Expt**$_3,i$.

**Proof.** We define several hybrid experiments. Experiment **G**$_0$ is identical to **Expt**$_3,i$.

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

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

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 $\text{sphf}$, and noting that $R_1 \neq S_1/pwd$ in instance $(i+1)$ (by construction of **Expt**$_3,i$).

In the next experiment **G**$_2$, the challenger generates the $S_1$ in the $(i+1)$-th instance as follows: $S_1 = pwd \cdot a^{r_1}$. That the view of $Z$ in experiments **G**$_1$ and **G**$_2$ are computationally indistinguishable follows from the DDH assumption in group $G_1$ (note $a$ is not being used by the challenger, now that Step 3(c) is no more).

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

**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 members.

At this point, the complete experiment **Expt**$_5$ can be described as follows:
1. The challenger $C$ picks the CRS just as in the real world. It retains $a, hk.$
2. On receiving $\text{NewSession}, sid, ssid, P_i, P_j, \text{pwd}, \text{role}$ from $Z, C$ generates $(HK_1, HP_1)$ by running $\text{ver.hkgen}(\text{crs}).$ Next, it generates $R_1, S_1, T_1$ by choosing $r_1$ at random, and setting $R_1 = g_1^r, S_1 = \text{pwd} \cdot a_1^r, T_1 = \text{sphf}.\text{pubH}(hp, \langle R_1, S_1/\text{pwd} \rangle, \iota_1),$ where $\iota_1 = H(sid, ssid, P_i, P_j, R_1, S_1, HP_1).$ It sends these values along with HP_1 to Z.
3. On receiving $R'_2, S'_2, T'_2, HP'_2$ from $Z,$ 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 $G_T$. 
   (b) Otherwise, if the message received is identical to message sent by $C$ in the same session (i.e. same ssid) on behalf of the peer, then output $sk_1 \leftarrow G_T$ (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 $\iota'_2 = H(sid, ssid, P_j, P_i, R'_2, S'_2, HP'_2).$
     if $T'_2 \neq \text{sphf}.\text{privH}(hk, \langle R'_2, S'_2/\text{pwd} \rangle, \iota'_2)$ then output randomly from $G_T.$
   (e) Else, compute $W_1 = \text{prover}(\text{crs}, \langle R_1, S_1/\text{pwd}, T_1, \iota_1 \rangle; r_1)$. Output $\text{ver.privH}(hk, \langle R'_2, S'_2/\text{pwd}, T_2, \iota'_2 \rangle) \cdot \text{ver.pubH}(HP'_2, W_1)$.
4. On a Corrupt call, if Step 2 has already happened then output $HK_1,$ pwd and $W_1 = \text{prover}(\text{crs}, \langle R_1, S_1/\text{pwd}, T_1, \iota_1 \rangle; r_1)$.

Handling Legitimate Messages

$\text{Expt}_6$ : 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 $C$ 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 $\text{Expt}_6$ is indistinguishable from $\text{Expt}_5$ 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 $\text{Expt}_{5,i}$ ($N \geq i \geq 1$), the Step 3(b) in the $i$-th temporally ordered instance is modified as required in $\text{Expt}_6$ description above. Experiment $\text{Expt}_{5,0}$ is same as experiment $\text{Expt}_5$, and experiment $\text{Expt}_{5,N}$ is same as experiment $\text{Expt}_6$.

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

Proof. The lemma is proved using several hybrid experiments of its own. The experiment $H_0$ is same as $\text{Expt}_{5,i-1}$. 

39
In experiment $H_1$, 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 $\eta$ output by the crs simulator). All proofs $W_1$ are still computed using the prover. Experiments $H_0$ and $H_1$ are indistinguishable as the QA-NIZK has the property that the simulated CRS and the real-world CRS are statistically identical.

In experiment $H_2$, in instance $i$, the value $W_1$ (in Step 3(e) or corruption) is generated using the proof simulator using trapdoor $\eta$. Indistinguishability follows by zero-knowledge property of the QA-NIZK as the proof being generated is on a language member.

In experiment $H_3$, in instance $i$, the value $T_1$ is generated using the private hash key $hk$, and the private hash function $sphf_{privH}$ (thus eliminating the use of witness $r_1$). Experiments $H_2$ and $H_3$ are indistinguishable by the correctness of $sphf$.

In experiment $H_4$, in instance $i$, the values $R_1$, $S_1$ are generated as $R_1 = g_1^{r_1}$, $S_1 = pwd \cdot a^{r_1} \cdot g_1^{r_1}$, where $r_1, r_1'$ are random and independent. This follows by employing DDH on $g_1, g_1^{r_1}, a$ and either $g_1^{ar_1}$ or $g_1^{ar_1+r_1}$.

In experiment $H_5$, in peer of instance $i$, in Step 3(d) the condition $T_2' \neq sphf_{privH}(hk, (R_2', S_2'/pwd), \iota_2')$ is replaced by “if $(S_2' \neq pwd \cdot (R_2')^\eta)$ or $T_2' \neq sphf_{privH}(hk, (R_2', S_2'/pwd), \iota_2')$”. Indistinguishability from experiment $H_4$ follows by smoothness property of the $sphf$, noting that at most one bad $sphf_{privH}$ is being output to the Adversary (namely $T_1$ in instance $i$).

In experiment $H_6$, in instance $i$, change Step 3(b) as follows: Step 3(b): Otherwise, if the message received is identical to message sent by $C$ 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 $\iota_2' = \mathcal{H}(sid, ssid, P_j, P_1, R_2', S_2'/pwd, hp_2')$, Output

$$ver_{privH}(hk_1, (R_2', S_2'/pwd, T_2'), \iota_2') \cdot ver_{privH}(hk_2, (R_1, S_1/pwd, T_1), \iota_1)$$

Here $hk_2$ is the $hk$ output by $ver_{hkgen}$ in the peer instance of instance $i$.

The experiments $H_6$ and $H_5$ are computationally indistinguishable by noting the following three facts:

1. In the peer of instance $i$ (which generated $hk_2$), in Step 3(e) the computation $ver_{privH}(hk_2, \cdot)$ 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 generated by $crs$.

3. Then, $ver_{privH}(hk_2, (R_1, S_1/pwd, T_1), \iota_1)$ is random even when the adversary is given $hp_2$ by smoothness of the QA-NIZK, since $S_1/pwd \neq R_1^\eta$. 40
In experiment $H_7$, in peer of instance $i$, in Step 3(d) the condition “if $(S'_2 \neq \text{pwd} \cdot (R'_2)^a)$ or $T'_2 \neq \text{spfh} \cdot \text{privH}(hk, (R'_2, S'_2/\text{pwd}), \iota'_2)$” is replaced by “if $T'_2 \neq \text{spfh} \cdot \text{privH}(hk, (R'_2, S'_2/\text{pwd}), \iota'_2)$”. Indistinguishability from experiment $H_6$ follows by smooth soundness of the QA-NIZK.

In experiment $H_8$, in instance $i$, $R_1$, $S_1$ are generated as $R_1 = g^{a^i}$, $S_1 = \text{pwd} \cdot a^i$, by employing DDH. In experiment $H_9$, in instance $i$, $T_1$ is generated using the public hash key $hp$, and witness $r_1$. Indistinguishability follows by correctness of the spfh.

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

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

- Else, compute $\iota'_2 = H(\text{sid}, \text{ssid}, P_i, P_1, R'_2, S'_2, \text{hp}', \text{privH})$.
  Compute $W_1 = \text{prover}(\text{crs}, (R_1, S_1/\text{pwd}, T_1, \iota_1), r_1)$. Output $\text{ver} \cdot \text{privH}(\text{hk}_1, (R'_2, S'_2/\text{pwd}, T'_2'), \iota'_2) \cdot \text{ver} \cdot \text{pubH}(\text{hp}', W_1)$

Indistinguishability follows by noting that $\text{hp}'$ is exactly the $\text{hp}$ 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 $(S'_2 \neq \text{pwd} \cdot (R'_2)^a)$ or $T'_2 \neq \text{spfh} \cdot \text{privH}(hk, (R'_2, S'_2/\text{pwd}), \iota'_2)$”. In other words, the disjunct $(S'_2 \neq \text{pwd} \cdot (R'_2)^a)$ 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 $(S'_2 \neq \text{pwd} \cdot (R'_2)^a)$ or $T'_2 \neq \text{spfh} \cdot \text{privH}(hk, (R'_2, S'_2/\text{pwd}), \iota'_2)$, then $R'_2, S'_2/\text{pwd}, T'_2$ and $\iota'_2$ 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 $S'_2 = \text{pwd} \cdot (R'_2)^a$, with witness $r_2$ for $R'_2$. Then, by correctness of the spfh,

$$\text{spfh} \cdot \text{privH}(hk, (R'_2, S'_2/\text{pwd}), \iota'_2) = \text{spfh} \cdot \text{pubH}(hp, (R'_2, S'_2/\text{pwd}), \iota'_2; r_2).$$

Then again, the tuple is not in the language.

Thus, ver $\cdot \text{privH}(\text{hk}_1, (R'_2, S'_2/\text{pwd}, T'_2), \iota'_2)$ is random, even when the Adversary is given $\text{hp}$, by smooth-soundness of the QA-NIZK.
**Expt₉**: In this experiment the Step 3(b) is dropped. In other words, the challenger code goes straight from 3(a) to 3(e).

Experiments **Expt₈** and **Expt₉** produce the same view for $Z$, since if both peers (of a instance) received legitimate messages forwarded by $Z$, then Step 3(e) computes the same instance key in both instances.

Finally, a simple examination shows that the view of $Z$ in **Expt₉** is identical to the real world protocol.