An extended abstract of this paper appears in the proceedings of the 29th International Conference on Selected Areas in Cryptography, SAC 2022. This is the full version.

PERKS: Persistent and Distributed Key Acquisition for Secure Storage from Passwords*

Gareth T. Davies[1] Jeroen Pijnenburg[2]

[1]Bergische Universität Wuppertal, Wuppertal, Germany
[2]Royal Holloway, University of London, Egham, United Kingdom

August 6, 2022

Abstract

We investigate how users of instant messaging (IM) services can acquire strong encryption keys to back up their messages and media with strong cryptographic guarantees. Many IM users regularly change their devices and use multiple devices simultaneously, ruling out any long-term secret storage. Extending the end-to-end encryption guarantees from just message communication to also incorporate backups has so far required either some trust in an IM or outsourced storage provider, or use of costly third-party encryption tools with unclear security guarantees. Recent works have proposed solutions for password-protected key material, however all require one or more servers to generate and/or store per-user information, inevitably invoking a cost to the users.

We define *distributed key acquisition* (DKA) as the primitive for the task at hand, where a user interacts with one or more servers to acquire a strong cryptographic key, and both user and server store as little as possible. We present a construction framework that we call PERKS—Password-based Establishment of Random Keys for Storage—providing efficient, modular and simple protocols that utilize Oblivious Pseudorandom Functions (OPRFs) in a distributed manner with minimal storage by the user (just the password) and servers (a single global key for all users). Along the way we introduce a formal treatment of DKA, and provide proofs of security for our constructions in their various flavours. Our approach enables key rotation by the OPRF servers, and for this we incorporate updatable encryption. Finally, we show how our constructions fit neatly with recent research on encrypted outsourced storage to provide strong security guarantees for the outsourced ciphertexts.

*Gareth T. Davies has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme, grant agreement 802823.

Contents

1 Introduction

Passwords are ubiquitous as authentication tokens and yet constructing schemes based on passwords is notoriously difficult to get right. Users regularly re-use and/or forget their passwords, application servers store passwords incorrectly, and more and more physical and technical tools are needed to prevent attacks and misuse. We consider the general problem of converting a human-memorable password into a single cryptographic secret, with minimal storage and communication requirements.

As a motivating case study, consider instant messaging (IM) applications: it is at present not clear what keying material should be used to encrypt messages and files that are stored on the user's device, and/or backed up to an external backup service or cloud storage provider (CSP)[1]. It is not clear what security properties are obtainable in the scenario where a user defends against potential loss of their device by backing up messages, never mind in each of the many other possible variations of the message storage scenario (long-term on-device encryption, temporary backup for 'device changeover', backup for immediate local deletion). Our solutions are targeted at this main scenario—where a user acquires a new phone and wants to recover their backed-up data using only their password—but with applications to the others[2]. In this setting a user may interact with their IM service, a CSP that stores messages

[1]In commercial settings there may exist on-premise file/backup storage, but in our more general case the entity storing the ciphertexts is regarded as external.

[2]We do not explicitly consider the scenario where the user has two devices in their possession and wishes to locally transfer messages and/or media from one device to another without the help of outsourced storage, as our approach would be overkill.

and media, and potentially other services that contribute to keying material: the user would prefer not to (fully) trust all of these services (or their device) and additionally would like to draw entropy from each of these services in deriving a(n initial) key for data encryption. We refer to all of these n entities, potentially including the IM and outsourced storage providers, as *key-contributing servers.* Our key tool is an oblivious PRF (OPRF): a user, holding a secret input x, engages in a protocol with a server, holding a key sk, where at the end of the protocol the user learns $\mathsf{F}(\mathrm{sk}, \mathrm{x})$ for some keyed pseudorandom function F and the server learns nothing.

A number of primitives exist in the literature that attempt to solve this problem and provide secure and distributed key generation, however all require the storage of user-specific information with the key servers. This is infeasible at the multi-billion-user scale required for modern IM applications, and would most likely result in this feature becoming a costly paid service. In particular, securely storing this per-user key material would often be done using a hardware security module (HSM), introducing significant key management challenges and financial costs. Further, many schemes require the user to generate the high-entropy secret in the first place and then securely distribute it, imposing a trust requirement on the client device and its randomness generation. We summarize these primitives in Sec. 1.3.

1.1 Problem Statement

There are at present (at least) three major stumbling blocks for deployment of encrypted backup systems for IM services:

- **Storage Cost.** Using existing techniques for OPRF-based password hardening would invoke per-user data to be stored at each of the key-contributing servers. For this reason it is necessary to minimize the storage burden for every entity in the system.
- **Key Longevity.** For the system to function, it is essential that over a long period of time the key acquisition interaction is 'long lived', in the sense that the key production operation must be deterministic and any secret values given as input to the operation (by user or server) must not change.
- **Trust Distribution.** In the IM setting it is by now industry standard to expect end-to-end-encryption (E2EE) of messages, ensuring that the storage server cannot decrypt sent content. With this in mind, it would appear risky to rely on the IM provider to act as a single key-contributing server, or to use only the IM server and cloud storage provider. It is thus desirable to introduce additional parties to the picture, and distribute the trust such that the user has fine-grained control of what they are able to do if they learn/believe that one or more of these parties has become compromised.

In our work we aim to overcome all of these challenges simultaneously, supplementing our proposals with rigorous security analyses.

The concept of key longevity is at odds with modern approaches to forward security that involve (regular) key rotation, and so any operation that does support key rotation must also enable the user to update their ciphertexts when this rotation occurs, which is a considerable technical challenge. In Sec. 5 we describe how key rotation can be done securely and efficiently within our framework.

1.2 Contributions

We design a novel formal framework for outsourced (and on-device) storage that allows a user to generate and store a cryptographic key with the aid of n key-contributing servers, with the constraints discussed already. We call the required primitive *distributed key acquisition* (DKA) and describe its syntax and security properties. We introduce a correctness game and two key indistinguishability games for DKA. We explain how a DKA scheme can be used in conjunction with encryption and key rotation to neatly solve the IM backup problem.

We provide two concrete constructions for DKA using OPRFs in a framework that we call PERKS (Password-based Establishment of Random Keys for Storage): the n out of n setting for when the user expects the key-contributing servers to be available for the lifetime of the system, and a threshold t out of n scheme based on secret sharing that tolerates $\mathrm{n} - \mathrm{t}$ servers being unavailable or compromised. Even in the event that $n - t + 1$ (or more, even all n) servers are corrupted by the same entity, all is not lost: this adversary must still perform an offline attack to recover the key. Our constructions are extremely simple but the analysis is certainly not: we introduce appropriate security properties for user and server privacy in our setting, and prove that our schemes—when instantiated using a variety of existing OPRFs with different features—meet the corresponding properties.

1.3 Related Primitives and Existing Literature

The existing primitive that is most closely related to our setting is (t, n) password-protected secret sharing (PPSS): a user that is already in possession of some secret value distributes it among n servers such that reconstruction is possible using only the password and interaction with $t + 1$ honest servers. Bagherzandi *et al.* [BJSL11] gave the first formal treatment and a scheme where the user needs to store/trust one or more public keys. Later schemes gave security in the presence of related passwords via the UC framework [CLN12], and in the setting where servers learn nothing in the reconstruction phase [CLLN14] (by sending out an encryption of a randomized quotient of the password and not the password itself).

Jarecki, Kiayias and Krawczyk [JKK14] gave threshold PPSS (and threshold PAKE) with optimal round complexity, but with the same setup assumptions as the prior papers. In [JKKX16] the same authors and Xu gained improved efficiency compared to previous work, and in essence these savings come from foregoing some heavy duty tools required to achieve the UC verifiability property of the OPRF. The same authors then presented TOPPSS [JKKX17] using a Threshold OPRF, where the secret sharing is performed on the level of the OPRF key, so derivation of the single OPRF output for a given input is an interactive protocol with $t + 1$ of the n servers. Abdalla *et al.* [ACNP16] defined robust PPSS, where a robust secret sharing scheme is used to detect cheating servers, also foregoing the need for a verifiable OPRF.

Password-Hardened Encryption (PHE) was introduced by Lai *et al.* [LER+18] and uses an oblivious external party for key derivation to protect against offline brute-force attacks on a ciphertext encrypted under just the password. Recognizing the single point of failure, Lai *et al.* introduced threshold PHE in [BEL+20]. The scheme requires a trusted third party to distribute all secret keys during initialization.

Other primitives exist that use a distributed OPRF service (i.e. the server stores, for each user, a sharing of an OPRF secret key) to derive a cryptographic secret, including Baum *et al.* [BFH+20] who use the OPRF to get a signature key pair for distributed single sign-on, and Das *et al.* [DHL22] who use a similar trick to obtain a signature key pair, then per-file encryption keys are created by computing a so-called extended POPRF for a second private input, namely a randomized hash of the file.

1.4 WhatsApp Encrypted Backup Rollout

In September 2021, WhatsApp announced [Wha21] that they would soon begin beta testing of an encrypted chat and media backup service that uses HSMs and the envelope part of the OPAQUE protocol [JKX18] in a manner that is conceptually similar to a $(1, 1)$-PPSS. In this subsection we discuss their system based on the details in the WhatsApp whitepaper and NCC Group's technical report [NG21] and explain the differences with our work.

OPAQUE is an asymmetric password-authenticated key exchange protocol that is a compiler of three components: an oblivious PRF to turn the user's password x into a strong secret value y, an 'envelope' mechanism whereby the user encrypts their AKE key material under y using symmetric encryption, and an AKE protocol. WhatsApp's approach uses the OPRF and a modified version of the envelope mechanism, but since no key exchange needs to occur the AKE component is dropped completely. In the WhatsApp system, at the point of registration (first ever backup), a client device generates a random 256-bit key k and then stores this as an encrypted record (envelope) in a 'HSM-based Backup Key Vault' so that it can later retrieve this key using only their password (PIN or passphrase): the HSM acts as the OPRF server and derives a per-user secret key sk_{uid} from a single master secret and uid when called. The envelope in the WhatsApp system is $\mathsf{PK.Enc}_{\mathrm{pk.HSM}}(\mathsf{SK.Enc}_{\mathrm{y}}(k))$, a public-key encryption of an encryption of k under the OPRF output value y. Later on when a user comes online to retrieve the contents of their envelope it is not apparent if this is sent encrypted under some user public key, and it would appear that this PKE scheme is not for protecting the channel, but rather so that the envelopes can be stored outside of the HSM. These envelopes are stored in an integrity-protected manner using a

Merkle tree. No security analysis of the system has been provided for the WhatsApp approach, and the only analysis available is the report by NCC Group [NG21] that does not discuss any formal security requirements for the system.

Intuitively, the WhatsApp approach relies on the tamper-resistant properties of the HSM to make sure that the OPRF key sk (that is used to derive sk_{uid}) is not leaked to any party. If this key is leaked, then an offline adversary can attempt to recover the file encryption key that is contained within the registration envelope. Our approach avoids assuming a HSM on the server side, and instead distributes the trust among a number of servers. An adversary in possession of a stolen client device needs to guess the correct password while avoiding WhatsApp's rate-limiting mechanisms, and thus performing this type of online attack is similar in our system.

Further, the WhatsApp system requires that the client device generates the user file encryption key k using 'a built-in cryptographically secure pseudorandom number generator', however as already stated, this is of no use if the device's randomness generation is already compromised during registration.

2 Preliminaries

2.1 Notation and Security Games

We specify scheme algorithms and security games in pseudocode. In such code we write '$var \leftarrow exp$' for evaluating expression *exp* and assigning the result to variable *var*. Here, expression *exp* may comprise the invocation of algorithms. If *var* is a set variable and *exp* evaluates to a set, we write $var \xleftarrow{\cup} exp$ shorthand for $var \leftarrow var \cup exp$. Similarly, if *var* is an integer variable and *exp* evaluates to an integer, we write $var \xleftarrow{+} exp$ shorthand for $var \leftarrow var + exp$. Associative arrays implement the 'dictionary' data structure: Once the instruction $A[\cdot] \leftarrow exp$ initialized all items of array A to the default value *exp*, with $A[idx] \leftarrow exp$ and $var \leftarrow A[idx]$ individual items indexed by expression *idx* can be updated or extracted. For a vector $\vec{v}$ we denote with $\mathrm{size}(\vec{v})$ the number of defined elements, i.e. elements that are not $\perp$, which may be less than the length of $\vec{v}$. Many algorithms take as input a security parameter λ, however for visual clarity we omit this wherever possible (further, this implicit representation is possible because we do not build any primitives from computational assumptions nor present any equational relationships that depend directly on λ).

Security games are parameterized by an adversary, and consist of a main game body plus zero or more oracle specifications. The adversary is allowed to call any oracle specified in the game at any time. The execution of a game starts with the main game body and terminates when a 'Stop with *exp*' instruction is reached, where the value of expression *exp* is taken as the outcome of the game. If the outcome of a game G is Boolean, we write $\Pr[\mathrm{G}(\mathcal{A})]$ for the probability that an execution of G with adversary $\mathcal{A}$ results in 1. We define macros for specific game-ending instructions: We write 'Win' for 'Stop with 1' and 'Lose' for 'Stop with 0', and for a condition C we write 'Require C' for 'If $\neg C$: Lose' and 'Reward C' for 'If C: Win'. We finally draw attention to an important detail of our algorithm and game notation: algorithms are allowed to abort. Here, by abort we mean the case where an algorithm does not generate output according to its syntax specification, but outputs some error indicator instead, e.g. outputs $\perp$. We have prefaced algorithms that may abort with the 'Try' statement. If an oracle calls an algorithm that aborts, the oracle also immediately aborts.

2.2 Secret Sharing Schemes

We now define a secret sharing scheme SSS, that allows an entity to share some secret value k among n parties, such that any t of the shares enable reconstruction of k, while any set of $\mathrm{t}-1$ shares reveals nothing about k. The exposition here is adapted from Boneh and Shoup [BS20].

A secret sharing scheme $\mathrm{SSS} = (\mathrm{SecShare}, \mathrm{SecCombine})$ over a finite set S_1 consists of two algorithms. Sharing algorithm $\mathrm{SecShare}(k, \mathrm{t}, \mathrm{n})$ is probabilistic, taking as input $k \in \mathrm{S}_1$ for $0 \leq \mathrm{t} \leq \mathrm{n}$ and returning shares $\vec{\alpha} = \{\alpha_1, \ldots, \alpha_\mathrm{n}\} \in \mathrm{S}_2^\mathrm{n}$. Reconstruction algorithm $\mathrm{SecCombine}(\vec{\alpha}')$ is deterministic, taking as input $\vec{\alpha}' = \{\alpha'_1, \ldots, \alpha'_\mathrm{t}\} \in \mathrm{S}_2^\mathrm{t}$ and returning the reconstructed secret k.

Correctness asks that for every secret $k \in \mathrm{S}_1$, every set of n shares $\vec{\alpha}$ output by $\mathrm{SecShare}(k, \mathrm{t}, \mathrm{n})$, and every subset $\{\alpha'_1, \ldots, \alpha'_\mathrm{t}\} = \vec{\alpha}' \subseteq \vec{\alpha}$ of size t, then $\mathrm{SecCombine}(\vec{\alpha}') = k$.

Definition 2.1 (SSS Security)**.** *A secret sharing scheme* (SecShare, SecCombine) *over* S_1 *is secure if for every* $k, k' \in \mathrm{S}_1$*, and every subset* $\vec{\alpha}' \in \mathrm{S}_2^{\mathrm{t}-1}$*, the distribution* $\mathrm{SecShare}(k, \mathrm{t}, \mathrm{n})[\vec{\alpha}']$ *is identical to the distribution* $\mathrm{SecShare}(k', \mathrm{t}, \mathrm{n})[\vec{\alpha}']$.

The most well known secret sharing scheme is due to Shamir [Sha79] using polynomial interpolation and is suitable for our purposes. The scheme is fully specified elsewhere [Sha79, BS20] and we refer to these sources for details. For the purposes of this paper, it is sufficient to know that Shamir's scheme is over $\mathrm{S}_1 = \mathbb{F}_q$ with prime power $q > \mathrm{n}$, where shares are elements of $\mathrm{S}_2 = \mathbb{F}_q^2$. We choose S_1 such that it matches our key space $\mathcal{K}$, for example $\mathrm{S}_1 = \mathbb{F}_{2^{256}}$ if we have a 256-bit key space.

2.3 OPRFs and their Variants

In the remaining subsections we describe some of the properties of oblivious PRFs in the literature and explain how they can be used in our protocols. OPRFs can be *verifiable* or not, and independently, *partially oblivious* or not, meaning there are four categories of OPRF that we consider.

Verifiability. Verifiable OPRFs (VOPRFs) require the server to commit to the secret key that it uses, and allow the user to verify that the correct operation was performed by the server with this committed key (in a way that does not reveal the key to the user). Syntactically, the server includes a proof in rep that the user can verify using a server public key pk.

Note that verifiability does not guarantee that a server uses the same key over multiple protocol runs. In order to check *key consistency*, the user is forced to store pk, and this value must be deterministically generated (this is the case for DH-based OPRFs where $\mathrm{pk} = g^{\mathrm{sk}}$). However, this storage need not be local: all users use the same pk so it is sufficient for this value to be published somewhere.

Partial Obliviousness. In many applications for OPRFs the server needs to partition the input space to reduce the impact of active attacks, and this is often done by choosing a different key for each user identity *uid*. In practice this could be done by applying some key derivation function to *uid* and sk before the protocol is run (see below for a short discussion of this approach). Partially-oblivious PRFs (POPRFs) contain a (plaintext) input t that provides automated partitioning, thus the server only needs one key for all users.

2.4 OPRF Literature

For a thorough treatment of OPRFs, see the SoK by Casacuberta *et al.* [CHL22]; here we summarize the most important literature for our approach. Oblivious PRFs were first formally defined by Freedman *et al.* [FIPR05]. A vast array of applications has arisen for OPRFs, including oblivious transfer and private set intersection [JL09], secure deduplication in cloud storage [KBR13], password-authenticated key exchange [JKX18], Cloudflare's anonymous authentication mechanism Privacy Pass [DGS+18], checking compromised credentials [TPY+19, LPA+19] and Meta's 'de-identified telemetry' scheme [HIJ+21].

The 2HashDH scheme by Jarecki *et al.* [JKK14] (detailed in Fig. 2) is very efficient and has been suggested for use in TLS 1.3 with OPAQUE as password-based authentication, and is subject to a standardization effort [DFHSW22, BKLW22, SKFB21].

There exist generic constructions of OPRFs from MPC techniques and homomorphic encryption that do not fit into the syntax in Sec. 2.5 since the communication does not follow a two-message pattern with the user sending the first message, see Section 2.4 of Casacuberta *et al.* [CHL22] for a summary. These constructions are generally useful for gaining properties that are not useful in our setting such as input batching [KKRT16] for amortized efficiency gains.

POPRFs. To our knowledge, the only two (explicit) POPRFs are those by Everspaugh *et al.* [ECS+15a] and Tyagi *et al.* [TCR+22], both of which are detailed in Fig. 2. The former requires a pairing which could be a hurdle in some practical applications, and the latter cannot support key rotation in a straightforward way.

Three works [JKR18, Leh19, DHL22] obtain partial-obliviousness for 2HashDH in a generic way by applying a PRF to the server key and public input and using that value as the per-user key. In our generic construction in Fig. 6 we use a similar idea to turn any of the two non-PO, DH-based schemes

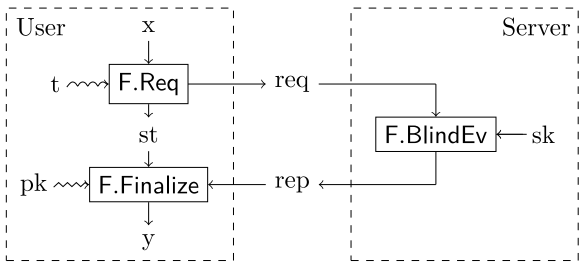

Figure 1: OPRF operation diagram. Public input t is only present in POPRFs and verification public key pk is only present in VOPRFs.

in Fig. 2 into partially-oblivious variants, with the additional benefit of efficient computation of per-user public keys in the verifiable setting.

The approach in the (unpublished) work of JKR18 [JKR18] actually works for any OPRF, and they present a non-updatable construction using 2HashDH and an updatable construction that uses HashDH, which is $\mathsf{H}_2(\mathsf{H}_1(\mathrm{x})^{\mathrm{sk}})$: this is not an OPRF since a user can use one interaction to obtain multiple evaluations.

Post-quantum OPRFs. Boneh *et al.* [BKW20] gave two constructions of OPRFs from isogenies: a VOPRF from SIDH [JD11] with a 'one-more' assumption and an OPRF from CSIDH [CLM+18]. A year later, Basso *et al.* [BKM+21] showed that the first construction's assumption does not hold and gave attacks on that OPRF; the second CSIDH-based scheme is unaffected by this work.

From lattices, Albrecht *et al.* [ADDS21] demonstrated that it is possible to build round-optimal (two messages in the online phase) VOPRFs from the Banerjee and Peikert PRF [BP14], but their protocols require large parameters and computation-heavy ZK proofs.

Kolesnikov *et al.* [KKRT16] sought to build multiple concurrent OPRF operations in a generic way from oblivious transfer (OT). OT can be built from post-quantum assumptions [PVW08, DvMN08], however the PRF functionality requires 5 communication rounds and is only 'relaxed' (as defined by Freedman *et al.* [FIPR05]). Note that these special purpose OPRFs, where more than two rounds are required, do not fit the syntax in Sec. 2.5.

Seres *et al.* [SHB21] have proposed an OPRF based on a version of the Legendre symbol problem and linked this hardness to multivariate quadratic equation systems.

2.5 Oblivious Pseudorandom Functions: Syntax

Following Everspaugh *et al.* [ECS+15a] and Tyagi *et al.* [TCR+22] we define OPRFs as a tuple $\mathsf{F} = (\mathsf{F.KG}, \mathsf{F.Req}, \mathsf{F.BlindEv}, \mathsf{F.Finalize}, \mathsf{F.Ev})$. The syntax captures two optional properties of OPRFs, namely partial obliviousness (user provides as input a public value in addition to its secret input, known as POPRFs) and verifiability (user is convinced that server has evaluated for a particular key, where the user learns a public commitment/representation of this key, known as VOPRFs), which both are useful but not essential in our constructions later on. In Sec. 2.3 we provide an overview of the prior work and how existing OPRF constructions can fit into our approach, with and without these additional properties. In Fig. 1 we depict the operation of a (P)OPRF with optional verifiability. Note that prior work including that of Tyagi *et al.* [TCR+22] has a parameter generation algorithm that passes some public parameters to all other algorithms. We choose to omit this detail for visual clarity.

$\mathsf{F.KG}$ generates a key pair $(\mathrm{pk}, \mathrm{sk})$ (or just sk for non-verifiable OPRFs), taking as input a security parameter. To form a request, the user runs $\mathsf{F.Req}(\mathrm{t}, \mathrm{x})$ with secret input x (and in POPRFs, public input t) and outputs a request req and a local state st. The server then runs $\mathsf{F.BlindEv}(\mathrm{sk}, \mathrm{t}, \mathrm{req})$ and outputs a response rep. The user finishes by running $\mathsf{F.Finalize}(\mathrm{pk}, \mathrm{rep}, \mathrm{st})$, either outputting the function evaluation y, or $\perp$ if it does not accept the outcome. The unblinded evaluation algorithm $\mathsf{F.Ev}(\mathrm{sk}, \mathrm{t}, \mathrm{x})$ outputs y or $\perp$. The server should not learn (anything about) the secret input x even after multiple interactions where x was provided as input, and the user should not learn (anything about) sk.

In Fig. 2 we detail the operation of three well-known OPRF protocols, all of which are reliant on Diffie-Hellman-like assumptions. In our constructions, the key-contributing servers will hold an OPRF secret key sk that they use for all users. User separation will either be done by employing a partially oblivious PRF with $\mathrm{t} = uid$, or by deriving a per-user key in each protocol invocation. We will always use the password as the user's secret input, so hereon $\mathrm{x} = pw$.

2.6 Oblivious Pseudorandom Functions: Security Notions

For *correctness* we require that honest OPRF evaluations consistently produce the same output when provided with equal inputs. We formalize this requirement in the correctness game in Fig. 3.

The privacy games capture that a server cannot glean any information about users' private inputs, nor link transcripts of requests/responses to OPRF output values, even with knowledge of the OPRF secret key. Our security notions are based on the security models of Tyagi *et al.* [TCR+22]. Their *user privacy* notion has two flavours:

- POPRIV-1 essentially models an honest but curious server and so does not require verifiability. In the game the adversary has a transcript-generation oracle that provides the entire transcript.
- POPRIV-2 models a malicious server by allowing the adversary to separately engage with oracles for OPRF request generation and OPRF output generation (in our notation denoted Challenge and Finalize respectively).

Additionally they gave a pseudorandomness game for *server privacy* where an adversary interacts with a (blinded) evaluation oracle and tries to distinguish genuine operation from operation with a random function.

Our PRIV-x games are adapted from the POPRIV-x games of Tyagi et al. [TCR+22]; in App. A we detail the differences: in short, PRIV-1 is a simplification of POPRIV-1 but with equivalent security, while PRIV-2 and POPRIV-2 are identical. We remark we only return the request req and not the response rep nor the final output value y in PRIV-1. It should be obvious that the adversary can compute rep on its own because it holds the secret key to evaluate $\mathsf{F.BlindEv}$, and in fact it can run this operation with arbitrary secret keys. Further, the adversary can run $\mathsf{F.Ev}$ on the values it has sent to Challenge using secret keys of its choosing, thus completing a full transcript. In PRIV-2 we do have a Finalize oracle, as the adversary is allowed to submit arbitrary responses and the correctness game makes no statement about this case. In App. A.2 we show that PRIV-1 and POPRIV-1 are equivalent. Furthermore, the $\mathsf{F.Req}$ algorithm no longer takes a public key as input and is thus independent of the server. Instead, the $\mathsf{F.Finalize}$ algorithm, which uses the public key to verify the response, now takes in the public key directly, rather than it being passed via request state.

Definition 2.2 (PRIV-x Security)**.** *The advantage of an adversary $\mathcal{A}$ in the* PRIV-x *security experiments defined in Fig. 3 for* $\mathrm{x} \in \{1, 2\}$ *and OPRF* F *is*

$$\mathbf{Adv}_{\mathsf{F}}^{\text{PRIV-x}}(\mathcal{A}) := \left|\Pr\left[\mathsf{G}_{\mathsf{F}}^{\text{PRIV-x}^1}(\mathcal{A}) = 1\right] - \Pr\left[\mathsf{G}_{\mathsf{F}}^{\text{PRIV-x}^0}(\mathcal{A}) = 1\right]\right|.$$

Our PRNG^b game is in essence the POPRF game in [TCR+22] for verifiable POPRFs, but extended to our multi-server setting: the oracles now return vectors instead of single elements. The game is parameterized by a simulator $\mathcal{S}$ and creates an environment for an adversary to interact with n OPRF servers via an oracle for function evaluation (Ev) and in modelling malicious clients a blinded evaluation oracle (BlindEv). Initially the game creates 2n server key pairs (or just secret keys for OPRFs that are not verifiable): in the $b = 0$ case the real function is used with one of the secret keys, and in the $b = 1$ case a random function is used and the simulator is tasked with providing appropriate responses but given

Name	POPRF?	Assumption	$\mathsf{F.Ev}(\mathrm{sk}, \mathrm{t}, \mathrm{x})$	req	rep
2HashDH [JKK14]	✗	OM-Gap-DH	$\mathsf{H}_2(\mathrm{x}, \mathsf{H}_1(\mathrm{x})^{\mathrm{sk}})$	$\mathsf{H}_1(\mathrm{x})^{\mathrm{r}}$	$\mathsf{H}_1(\mathrm{x})^{\mathrm{r}\cdot\mathrm{sk}}$
Pythia [ECS+15a]	✓	OM-BCDH	$e(\mathsf{H}_3(\mathrm{t}), \mathsf{H}_1(\mathrm{x}))^{\mathrm{sk}}$	$\mathsf{H}_1(\mathrm{x})^{\mathrm{r}}, \mathrm{t}$	$e(\mathsf{H}_3(\mathrm{t}), \mathsf{H}_1(\mathrm{x})^{\mathrm{r}})^{\mathrm{sk}}$
3HashSDHI [TCR+22]	✓	OM-Gap-SDHI	$\mathsf{H}_2(\mathrm{t}, \mathrm{x}, \mathsf{H}_1(\mathrm{x})^{\frac{1}{\mathsf{H}_3(\mathrm{t})+\mathrm{sk}}})$	$\mathsf{H}_1(\mathrm{x})^{\mathrm{r}}, \mathrm{t}$	$\mathsf{H}_1(\mathrm{x})^{\frac{\mathrm{r}}{\mathsf{H}_3(\mathrm{t})+\mathrm{sk}}}$

Figure 2: Comparison of selected (partially) oblivious PRFs from the literature. Hash functions H_i are labelled for comparison purposes, but when used in protocols the domains and ranges will be different.

```
**Game** CORR($\mathcal{A}, n$)
$Q[\cdot] \leftarrow \emptyset$
For $i \in [1\,..\,n]$:
    $(\mathrm{pk}^i, \mathrm{sk}^i) \leftarrow_{\$} \mathsf{F.KG}$
$\mathcal{A}(\vec{\mathrm{pk}}, \vec{\mathrm{sk}})$
Lose

**Oracle** Protocol(t, x)
$(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(\mathrm{t}, \mathrm{x})$
For $i \in [1\,..\,n]$:
    $\mathrm{rep}^i \leftarrow \mathsf{F.BlindEv}(\mathrm{sk}^i, \mathrm{t}, \mathrm{req})$
    $\mathrm{y}^i \leftarrow \mathsf{F.Finalize}(\mathrm{pk}^i, \mathrm{rep}^i, \mathrm{st})$
$Q[\mathrm{t}, \mathrm{x}] \overset{\cup}{\leftarrow} \{\vec{\mathrm{y}}\}$
Reward $|Q[\mathrm{t}, \mathrm{x}]| > 1$
Return $(\vec{\mathrm{y}}, \mathrm{req}, \vec{\mathrm{rep}})$

**Game** PRIV-2$^b(\mathcal{A})$
$i \leftarrow 0$
$\mathrm{ST}[\cdot] \leftarrow \perp$
$b' \leftarrow \mathcal{A}()$
Stop with $b'$

**Oracle** Challenge($\mathrm{t}, \mathrm{x}_0, \mathrm{x}_1$)
$i \overset{+}{\leftarrow} 1$
$(\mathrm{req}_0, \mathrm{st}_0) \leftarrow \mathsf{F.Req}(\mathrm{t}, \mathrm{x}_0)$
$(\mathrm{req}_1, \mathrm{st}_1) \leftarrow \mathsf{F.Req}(\mathrm{t}, \mathrm{x}_1)$
$\mathrm{ST}[i] \leftarrow (\mathrm{st}_0, \mathrm{st}_1)$
Return $(i, \mathrm{req}_b, \mathrm{req}_{1-b})$

**Oracle** Finalize($j, \mathrm{pk}, \mathrm{rep}, \mathrm{rep}'$)
Require $j \in [1\,..\,i]$
$(\mathrm{st}_0, \mathrm{st}_1) \leftarrow \mathrm{ST}[j]$
$\mathrm{y}_b \leftarrow \mathsf{F.Finalize}(\mathrm{pk}, \mathrm{rep}, \mathrm{st}_b)$
$\mathrm{y}_{1-b} \leftarrow \mathsf{F.Finalize}(\mathrm{pk}, \mathrm{rep}', \mathrm{st}_{1-b})$
If $\mathrm{y}_0 = \perp$ or $\mathrm{y}_1 = \perp$:
    Return $(\perp, \perp)$
Return $(\mathrm{y}_0, \mathrm{y}_1)$
```

```
**Game** PRIV-1$^b(\mathcal{A})$
$b' \leftarrow \mathcal{A}()$
Stop with $b'$

**Oracle** Challenge($\mathrm{t}, \mathrm{x}_0, \mathrm{x}_1$)
$(\mathrm{req}_0, \mathrm{st}_0) \leftarrow \mathsf{F.Req}(\mathrm{t}, \mathrm{x}_0)$
$(\mathrm{req}_1, \mathrm{st}_1) \leftarrow \mathsf{F.Req}(\mathrm{t}, \mathrm{x}_1)$
Return $(\mathrm{req}_b, \mathrm{req}_{1-b})$

**Game** PRNG$^b(\mathcal{A}, n)$
$\mathrm{BE}[\cdot] \leftarrow 0$; $\mathrm{RE}[\cdot] \leftarrow 0$
For $i \in [1\,..\,n]$:
    $(\mathrm{pk}_0^i, \mathrm{sk}_0^i) \leftarrow_{\$} \mathsf{F.KG}$
    $(\mathrm{pk}_1^i, \mathrm{sk}_1^i) \leftarrow_{\$} \mathsf{F.KG}$
    $G^i \leftarrow_{\$} \{g \mid g : \mathcal{U} \times \mathcal{I} \to \mathcal{O}\}$
$b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}}_b)$
Stop with $b'$

**Oracle** Ev(t, x)
For $i \in [1\,..\,n]$:
    $\mathrm{y}_0 \leftarrow \mathsf{F.Ev}(\mathrm{sk}_0^i, \mathrm{t}, \mathrm{x})$
    $\mathrm{y}_1 \leftarrow G^i(\mathrm{t}, \mathrm{x})$
Return $\vec{\mathrm{y}}_b$

**Oracle** BlindEv(t, req)
$\mathrm{BE}[\mathrm{t}] \overset{+}{\leftarrow} 1$
For $i \in [1\,..\,n]$:
    $\mathrm{rep}_0^i \leftarrow \mathsf{F.BlindEv}(\mathrm{sk}_0^i, \mathrm{t}, \mathrm{req})$
    $\mathrm{rep}_1^i \leftarrow \mathcal{S}.\mathsf{BlindEv}(\mathrm{sk}_1^i, \mathrm{t}, \mathrm{req})$
Return $\vec{\mathrm{rep}}_b$

**Oracle** $H(x)$
$h_0 \leftarrow \mathsf{RO}(x)$
$h_1 \leftarrow \mathcal{S}.\mathsf{Ev}(x)$
Return $h_b$

$\mathcal{S}$-**Oracle** RestrictedEv(t, x)
Require $\mathrm{RE}[\mathrm{t}] < \mathrm{BE}[\mathrm{t}]$
$\mathrm{RE}[\mathrm{t}] \overset{+}{\leftarrow} 1$
$\vec{\mathrm{y}} \leftarrow \mathrm{Ev}(\mathrm{t}, \mathrm{x})$
Return $\vec{\mathrm{y}}$
```

Figure 3: OPRF Games for the multiple servers setting. For the meaning of instructions Stop with, Lose, Reward, and Require see Sec. 2.

the other secret key. The Ev oracle returns either the output of the F.Ev algorithm or a random function. The adversary also has access to the BlindEv oracle, which either returns the output of the F.BlindEv algorithm or the response generated by the simulator $\mathcal{S}$.BlindEv. Queries to oracle H are either answered by a random oracle query or simulated by $\mathcal{S}$.Ev. Crucially, to maintain consistency between Ev and BlindEv queries, the simulator can obtain Ev outputs via its own $\mathcal{S}$-Oracle RestrictedEv. However, the simulator is restricted: the number of queries to RestrictedEv is bounded by the number of adversary queries to BlindEv, specific for each public input. This ensures that the adversary cannot compute more POPRF evaluations than the number of oracle queries it made. Moreover, restricting per public input means that querying with public input t_1 cannot help the adversary compute the evaluation for another public input $\mathrm{t}_2 \neq \mathrm{t}_1$. For a more detailed description of (the single server version of) this game, see Section 3 of [TCR+22]: there are many subtleties regarding the simulator and limited evaluation however these are not required for our purposes, since we only build from secure (P)OPRFs.

Definition 2.3 (PRNG Security)**.** *The advantage of an adversary $\mathcal{A}$ in the* PRNG *security experiments* *defined in Fig. 3 for OPRF* F *is*

$$\mathbf{Adv}^{\mathrm{PRNG}}_{\mathsf{F},\mathcal{S}}(\mathcal{A},\mathrm{n}) := \left|\Pr\left[\mathsf{G}^{\mathrm{PRNG}^1}_{\mathsf{F},\mathcal{S}}(\mathcal{A},\mathrm{n})=1\right] - \Pr\left[\mathsf{G}^{\mathrm{PRNG}^0}_{\mathsf{F},\mathcal{S}}(\mathcal{A},\mathrm{n})=1\right]\right|.$$

3 DKA and Security Models

In this section we formally define the syntax of a distributed key acquisition scheme and define security via two games for key indistinguishability. The weaker KIND-1 security game effectively models an honest but curious adversary as it may call oracles for honest protocol executions for a user to learn its requests and responses. In the KIND-2 security game the adversary is in complete control of the servers and may choose arbitrarily how to respond to user requests.

3.1 Distributed Key Acquisition

A distributed key acquisition scheme is an interactive protocol between parties, where the parties can either be users or servers. Let S be a set of n servers that are (initially) available to users. Let $uid \in \mathcal{UID}$ be a user identity that (initially) has a password $pw \in \mathcal{D}$ in some dictionary $\mathcal{D}$.

The system is initialized by gen(n) which assigns a key pair to each server $\mathrm{S}_i \in \mathrm{S}$, and public keys are then distributed to users. A user initializes itself in the system and acquires a key k and possibly some setup values $\vec{\mathrm{SV}}$ by running $\mathrm{init}(uid, pw, \vec{\mathrm{pk}}, \mathrm{S})$. We remark that $\vec{\mathrm{SV}}$ is not secret, so it can be stored alongside the backed up data. The init procedure sends out a request req to each server, who will run $\mathsf{server.op}(\mathrm{sk}_i, uid, \mathrm{req})$ to respond. An individual user's interaction with the system is defined by a threshold $\mathrm{t} \in [1\,..\,\mathrm{n}]$ which is the number of servers that are required to be honest and available in order for the user to reconstruct their secret. Later, a user can acquire the OPRF output values $\vec{\mathrm{y}}$ by running $\mathrm{acquire}(uid, pw, \vec{\mathrm{pk}}, \mathrm{S})$ and recover their key by subsequently running $\mathrm{recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$. Syntactically this takes as input all OPRF output values, but some may not be set, i.e. y_i may be $\perp$ if server S_i did not respond. With C, the set of chosen servers, a user indicates which OPRF output values to use for the key recovery. We assume that passwords are selected uniformly at random from the set $\mathcal{D}$ throughout the rest of this paper. However, similarly to the game-based PAKE literature, it is possible to cast the choosing of passwords according to (the min-entropy of) some distribution Dist [BCP04, BP13].

3.2 A Unified Security Notion for DKA

We first describe the correctness game for DKA schemes depicted in Fig. 4. The correctness game initializes the secret and public keys for all servers and initializes several game variables to keep track of the game state. The adversary controls the honest executions of the protocol via its oracles Init, Acquire and Recover. It has complete control over the inputs, specifying which user identity uid, password pw and public keys $\vec{\mathrm{pk}}$ to use in Init and Acquire, and the set of chosen servers C (for reconstruction) and setup values $\vec{\mathrm{SV}}$ in the Recover oracle. The counter r is a game variable to associate the Recover query with the corresponding Acquire query. This gives the adversary more freedom as we do not require these oracles to be used in succession. Via the Corrupt oracle the adversary is allowed to corrupt up to $t-1$ servers. Recall that the oracle immediately aborts if a procedure prefaced by 'Try' aborts. In particular, a correct construction can abort if it is fed garbage input (i.e. an empty set of chosen servers) as otherwise the adversary would trigger the 'Reward' line that wins the correctness game. The adversary wins the correctness game if it manages to create two different keys for a set of user id, password and setup values.

Next, we describe the security games for DKA schemes, KIND-x for $\mathrm{x} \in \{1,2\}$, provided in Fig. 5, which ultimately capture key indistinguishability: The task of the adversary is to distinguish real keys generated by the protocol from random. We remark that this implies other security properties such as privacy of the user's password: if the adversary learns (information about) a user's password it can compute the real key and compare this to the real or random key from the Challenge oracle to gain an advantage. Similarly to PRIV-x, the KIND-x game comes in two flavors: $\mathrm{x}=1$ corresponds to an adversary that may compromise servers but will subsequently follow the protocol honestly, and $\mathrm{x}=2$ which allows arbitrary server behaviour and thus intuitively security in this setting will require verifiable responses from the servers.

Initially, the game assigns passwords to all users in the user identity space $\mathcal{UID}$. An adversary can observe network traffic for executions of the protocol via its oracles Init and Acquire, and it specifies the

```
**Game** $\mathrm{CORR}(\mathcal{A}, \mathrm{t}, \mathrm{n})$
$\mathrm{K}[\cdot] \leftarrow \emptyset$
$\mathrm{CRED}[\cdot] \leftarrow \bot$
$\mathrm{Y}[\cdot] \leftarrow \bot$
$\mathrm{CO} \leftarrow \emptyset$
$r \leftarrow 0$
$(\vec{\mathrm{sk}}, \vec{\mathrm{pk}}) \leftarrow \mathrm{gen}(\mathrm{n})$
$\mathrm{S} \leftarrow \mathrm{init}(\vec{\mathrm{sk}})$
$\mathcal{A}(\vec{\mathrm{pk}})$
Lose
```

```
**Oracle** $\mathrm{Init}(uid, pw, \vec{\mathrm{pk}})$
Try: $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{init}(uid, pw, \vec{\mathrm{pk}}, \mathrm{S})$
$\mathrm{K}[pw, uid, \vec{\mathrm{SV}}] \overset{\cup}{\leftarrow} \{k\}$
$(\mathrm{req}, \vec{\mathrm{rep}}) \leftarrow \mathrm{Transcript}(\mathrm{S})$
Return $(k, \mathrm{req}, \vec{\mathrm{rep}}, \vec{\mathrm{SV}})$

**Oracle** $\mathrm{Acquire}(uid, pw, \vec{\mathrm{pk}})$
$r \overset{+}{\leftarrow} 1$
$\mathrm{CRED}[r] \leftarrow (uid, pw)$
$\mathrm{Y}[r] \leftarrow \mathrm{acquire}(uid, pw, \vec{\mathrm{pk}}, \mathrm{S})$
$(\mathrm{req}, \vec{\mathrm{rep}}) \leftarrow \mathrm{Transcript}(\mathrm{S})$
Return $(\mathrm{req}, \vec{\mathrm{rep}}, r)$
```

```
**Oracle** $\mathrm{Corrupt}(i)$
$\mathrm{CO} \overset{\cup}{\leftarrow} \{i\}$
Require $|\mathrm{CO}| < \mathrm{t}$
Return $\mathrm{sk}_i$

**Oracle** $\mathrm{Recover}(r, \mathrm{C}, \vec{\mathrm{SV}})$
$(uid, pw) \leftarrow \mathrm{CRED}[r]$
$\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
Try: $k \leftarrow \mathrm{recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
$\mathrm{K}[pw, uid, \vec{\mathrm{SV}}] \overset{\cup}{\leftarrow} \{k\}$
Reward $|\mathrm{K}[pw, uid, \vec{\mathrm{SV}}]| > 1$
Return $k$
```

Figure 4: Correctness game for DKA with algorithms gen, init, acquire and recover. Transcript is a special game procedure that records network requests sent by the DKA algorithms. For the meaning of instructions Lose, Reward, and Require see Sec. 2.

user identity for these protocol runs. In the KIND-1 game for honest executions of the protocol, this is modelled by providing the adversary a transcript of the network requests to the servers S, handled by the game. For the KIND-2 game, the network requests are sent to the adversary directly, so the game does not need to record the transcript. The adversary may respond in any way it likes, in particular it may respond honestly. To aid the adversary in responding honestly without requiring it to corrupt a server, it can query the BlindEv oracle, which will return the honest response rep. Effectively, in the KIND-2 game the adversary becomes an active man-in-the-middle between the Init and Acquire oracles (representing the user) and the BlindEv oracle (representing the server).

We split the key reconstruction procedure into two processes to model the online communication (Acquire) between the user and the servers, and the key calculation done locally (Recover) by a user based on the servers it chooses to utilize and the public setup values. We allow the adversary to specify these setup values $\vec{\mathrm{SV}}$ since the system's security should not rely on them being secret nor authentic. To model online attacks, i.e. login attempts for specific users, the adversary can call Reveal with a purported password and user identity, and if this guess is correct it receives the file encryption key for that user; if incorrect it receives nothing (this mimics the subsequent inability to decrypt files).

Definition 3.1 (KIND-x Security)**.** *The advantage of an adversary $\mathcal{A}$ in the* KIND-x *games defined in Fig. 5 for* $\mathrm{x} \in \{1, 2\}$ *and distributed key acquisition scheme* DKA *is*

$$\mathbf{Adv}^{\text{KIND-x}}_{\mathsf{DKA}}(\mathcal{A}) := \left|\Pr\left[\mathsf{G}^{\text{KIND-x}^1}_{\mathsf{DKA}}(\mathcal{A}) = 1\right] - \Pr\left[\mathsf{G}^{\text{KIND-x}^0}_{\mathsf{DKA}}(\mathcal{A}) = 1\right]\right|.$$

As is natural in the password setting, it is necessary to consider the fact that passwords could be guessable and a successful guess that occurs before any rate-limiting has kicked in will result in an adversary compromising a particular user. The advantage statement needs to take into account the following generic attack, where the queries are all for a single user identity *uid*: the adversary runs Init, then makes q queries to Reveal for randomly chosen passwords in the password space, then queries Challenge. If any of the Reveal queries returned something other than $\bot$, then a user key was set for that user identity, and if this value is equal to the key provided by Challenge then the adversary outputs 0, and if it's different it outputs 1 (if it received $\bot$ for all Reveal queries then it just guesses). As a result, we regard a DKA scheme as being secure if

$$\mathbf{Adv}^{\text{KIND-x}}_{\mathsf{DKA}}(\mathcal{A}) \leq \mathcal{O}\left(\frac{\mathrm{q}}{|\mathcal{D}|}\right) + \delta,$$

where q is the number of queries made by the adversary to the Reveal oracle in the course of the experiment, $|\mathcal{D}|$ is the size of the password dictionary and δ is some negligible function in the security parameter λ.

```
Game KIND-1^b(A, t, n)
    $\mathrm{K}[\cdot] \leftarrow \times;\ r \leftarrow 0$
    $\mathrm{PW}[\cdot] \leftarrow \perp;\ \mathrm{Y}[\cdot] \leftarrow \perp$
    For $uid \in \mathcal{UID}$:
        $\mathrm{PW}[uid] \leftarrow_\$ \mathcal{D}$
    $\mathrm{CH} \leftarrow \emptyset;\ \mathrm{CO} \leftarrow \emptyset$
    $(\vec{\mathrm{sk}}, \vec{\mathrm{pk}}) \leftarrow \mathrm{gen}(\mathrm{n})$
    $\mathrm{S} \leftarrow \mathrm{init}(\vec{\mathrm{sk}})$
    $b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
    Stop with $b'$

Oracle Init($uid, \vec{\mathrm{pk}}$)
    $pw \leftarrow \mathrm{PW}[uid]$
    Try:
        $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{init}(uid, pw, \vec{\mathrm{pk}}, \mathrm{S})$
    $\mathrm{K}[pw, uid] \leftarrow k$
    $(\mathrm{req}, \vec{\mathrm{rep}}) \leftarrow \mathrm{Transcript}(\mathrm{S})$
    Return $(\mathrm{req}, \vec{\mathrm{rep}}, \vec{\mathrm{SV}})$

Oracle Acquire($uid, \vec{\mathrm{pk}}$)
    $r \overset{+}{\leftarrow} 1$
    $pw \leftarrow \mathrm{PW}[uid]$
    $\mathrm{Y}[r] \leftarrow \mathrm{acquire}(uid, pw, \vec{\mathrm{pk}}, \mathrm{S})$
    $(\mathrm{req}, \vec{\mathrm{rep}}) \leftarrow \mathrm{Transcript}(\mathrm{S})$
    Return $(\mathrm{req}, \vec{\mathrm{rep}}, r)$
```

```
Oracle Recover($r, \mathrm{C}, \vec{\mathrm{SV}}$)
    $\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
    Try:
        $k \leftarrow \mathrm{recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
    $\mathrm{K}[pw, uid] \leftarrow k$
    Return

Oracle Reveal($pw', uid$)
    $k \leftarrow \mathrm{K}[pw', uid]$
    Return $k$

Oracle Challenge($uid$)
    $pw \leftarrow \mathrm{PW}[uid]$
    Require $\mathrm{K}[pw, uid] \neq \times$
    Require $uid \notin \mathrm{CH}$
    $\mathrm{CH} \overset{\cup}{\leftarrow} \{uid\}$
    $k_0 \leftarrow \mathrm{K}[pw, uid]$
    $k_1 \leftarrow_\$ \mathcal{K}$
    Return $k_b$

Oracle Corrupt($i$)
    $\mathrm{CO} \overset{\cup}{\leftarrow} \{i\}$
    Require $|\mathrm{CO}| < \mathrm{t}$
    Return $\mathrm{sk}_i$
```

```
Game KIND-2^b(A, t, n)
    $\mathrm{K}[\cdot] \leftarrow \times;\ r \leftarrow 0$
    $\mathrm{PW}[\cdot] \leftarrow \perp\ \mathrm{Y}[\cdot] \leftarrow \perp$
    For $uid \in \mathcal{UID}$:
        $\mathrm{PW}[uid] \leftarrow_\$ \mathcal{D}$
    $\mathrm{CH} \leftarrow \emptyset;\ \mathrm{CO} \leftarrow \emptyset$
    $(\vec{\mathrm{sk}}, \vec{\mathrm{pk}}) \leftarrow \mathrm{gen}(\mathrm{n})$
    $b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
    Stop with $b'$

Oracle BlindEv($i, uid, \mathrm{req}$)
    $\mathrm{rep}_i \leftarrow \mathsf{server.op}(\mathrm{sk}_i, uid, \mathrm{req})$
    Return $\mathrm{rep}_i$

Oracle Init($uid, \vec{\mathrm{pk}}$)
    $pw \leftarrow \mathrm{PW}[uid]$
    Try:
        $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{init}(uid, pw, \vec{\mathrm{pk}}, \mathcal{A})$
    $\mathrm{K}[pw, uid] \leftarrow k$
    Return $\vec{\mathrm{SV}}$

Oracle Acquire($uid, \vec{\mathrm{pk}}$)
    $r \overset{+}{\leftarrow} 1$
    $pw \leftarrow \mathrm{PW}[uid]$
    $\mathrm{Y}[r] \leftarrow \mathrm{acquire}(uid, pw, \vec{\mathrm{pk}}, \mathcal{A})$
    Return $r$
```

Figure 5: Key indistinguishability games for DKA with algorithms gen, init, acquire, recover and server.op. The oracles in the middle column are equal for both games and hence only displayed once. Transcript is a special game procedure that records network requests sent by the DKA algorithms. Assuming $\times \notin \mathcal{K}$, we encode uninitialized keys with $\times$. For the meaning of instructions Stop with and Require see Sec. 2.

4 Constructions

In this section we present two schemes and prove their security in the models from Sec. 3. Our constructions are parameterized by an Oblivious PRF F = (F.KG, F.Req, F.BlindEv, F.Finalize, F.Ev) and follow a generic blueprint, this blueprint is portrayed in Fig. 6.

4.1 Generic Construction

There are four possibilities for OPRFs: verifiable or non-verifiable, and partially-oblivious or regular. In order to handle both partially-oblivious and regular oblivious PRFs, we desire that each server can derive a per-user key on the fly, see the Cli.SKG and Cli.PKG algorithms in Fig. 6. If the OPRF is partially oblivious then the auxiliary input *uid* creates domain separation in the key used by the OPRF server, and so the per-user key pair is just the single key pair created by F.KG, for all users. To work with verifiable (regular) OPRFs such that the servers are not required to store per-user data, we need the OPRF secret key sk to be a group element with public key g^{sk} for some generator g. Then, the server simply multiplies in the group its own OPRF secret key with a hash of the user's identity to create the secret key component, and raises its own public key to the hash value to get the public key component; this public key component is provided to the user. If a verifiable OPRF is being used and it does not have this method of operation, and this includes all OPRFs built from non-DH assumptions, then another mechanism is required. For non-verifiable (non-DH) OPRFs, the server simply needs some way of generating per-user secret keys using its master secret key and the user's identity, e.g. a key derivation function.

The init algorithm allows the user to compute a random key using its password *pw* and a set of servers S that can be reconstructed later using *pw* and (a subset of) S. It creates an OPRF request and awaits the response for each server. We have used the 'Await' keyword in Fig. 6 to indicate this computation

```
Proc gen(n)
  For $i \in [1..n]$:
    $(\mathrm{sk}_i, \mathrm{pk}_i) \leftarrow \mathsf{F.KG}$
  Return $\vec{\mathrm{sk}}, \vec{\mathrm{pk}}$

Proc Cli.SKG($\mathrm{sk}_i$, $uid$)
● $csk_i \leftarrow \mathrm{sk}_i \cdot \mathsf{H}_{\mathsf{Cli.KG}}(uid)$
○ $csk_i \leftarrow \mathrm{sk}_i$
  Return $csk_i$

Proc Cli.PKG($\mathrm{pk}_i$, $uid$)
● $cpk_i \leftarrow \mathrm{pk}_i^{\mathsf{H}_{\mathsf{Cli.KG}}(uid)}$
○ $cpk_i \leftarrow \mathrm{pk}_i$
  Return $cpk_i$

Proc server.op($\mathrm{sk}_i$, $uid$, req)
  $csk_i \leftarrow \mathsf{Cli.SKG}(\mathrm{sk}_i, uid)$
  $\mathrm{rep}_i \leftarrow \mathsf{F.BlindEv}(csk_i, uid, \mathrm{req})$
  Return $\mathrm{rep}_i$

Proc init($uid$, $pw$, $\vec{\mathrm{pk}}$, S)
  $(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw)$
  For $i \in [1..n]$:
    Await $\mathrm{rep}_i \leftarrow \mathsf{server.op}(\mathrm{S}_i, uid, \mathrm{req})$
    $cpk_i \leftarrow \mathsf{Cli.PKG}(\mathrm{pk}_i, uid)$
    $\mathrm{y}_i \leftarrow \mathsf{F.Finalize}(cpk_i, \mathrm{rep}_i, \mathrm{st})$
  Try: $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{setup}(\vec{\mathrm{y}})$
  Return $k, \vec{\mathrm{SV}}$

Proc acquire($uid$, $pw$, $\vec{\mathrm{pk}}$, S)
  $(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw)$
  For $i \in [1..n]$:
    Await $\mathrm{rep}_i \leftarrow \mathsf{server.op}(\mathrm{S}_i, uid, \mathrm{req})$
    $cpk_i \leftarrow \mathsf{Cli.PKG}(\mathrm{pk}_i, uid)$
    $\mathrm{y}_i \leftarrow \mathsf{F.Finalize}(cpk_i, \mathrm{rep}_i, \mathrm{st})$
  Return $\vec{\mathrm{y}}$

Proc recover($\vec{\mathrm{y}}$, C, $\vec{\mathrm{SV}}$)
  For $i \in [1..n] \setminus \mathrm{C}$:   $\mathrm{y}_i \leftarrow \perp$
  Try:   $k \leftarrow \mathrm{reconstruct}(\vec{\mathrm{y}}, \vec{\mathrm{SV}})$
  Return $k$
```

Figure 6: PERKS, a generic DKA protocol construction. The lines marked with ● are executed iff a standard OPRF is used as building block. The lines marked with ○ are executed iff a POPRF is used as building block. Procedures setup and reconstruct are as in Fig. 7 for the n out of n setting and as in Fig. 8 for the t out of n setting. In a slight abuse of notation, we specify server.op on the user side with a server S_i as input, who will use its secret key sk_i to evaluate the procedure.

is not done locally. Computing the OPRF responses is done by server.op, which is a wrapper of the OPRF's blind evaluation function using the per-user key. The responses are finalized by init to obtain the OPRF output values, which are used by setup to compute the key and potentially some setup values. The setup algorithm is setting specific and will be discussed later.

The reconstruction of the key is similar, but split in two algorithms acquire and recover. This allows the user to choose which subset of servers C to use, after seeing the OPRF outputs (some servers may not respond). Indeed, this modularization allows any choice function from the OPRF output space to the power set of S. The OPRF output values are computed by acquire, to be used subsequently by the local reconstruct algorithm inside recover to recompute the key. Similarly to setup, the reconstruct algorithm is setting specific.

We remark that the scheme stops working (in the sense that the user cannot decrypt their files) if one of the servers chosen for recover is not consistent with its responses. If the OPRF is verifiable, then the user can identify which server has replied inconsistently and exclude it from the servers chosen for recover, so it is recommended to use verifiable OPRFs when available. Alternatively, assuming the set of servers is small, the user could proceed by trying different subsets and rerunning recover until successful. In Sec. 4.5 we discuss how existing OPRF schemes from the literature can be used in PERKS.

4.2 n out of n setting

In Fig. 7 we provide the construction for the setting where all n key servers are required for key (re)production. The setup values $\vec{\mathrm{SV}}$ are effectively ignored in this setting, they are only present in the construction to be syntactically correct. The user derives their key as the XOR of the OPRF output values. The construction allows a user to generate a key even if it cannot produce randomness itself, and as long as at least one of the n servers is not malicious, the key produced will be pseudorandom.

4.3 t out of n setting

We now demonstrate how to use secret sharing to derive a key using only a subset of the active servers. In addition to OPRF F, the protocol uses a secret sharing scheme SSS = (SecShare, SecCombine). The

```
Proc setup($\vec{y}$)
00 Assert size($\vec{y}$) = n
01 $k \leftarrow y_1 \oplus \ldots \oplus y_n$
02 Return $(k, \perp)$

Proc reconstruct($\vec{y}, \vec{SV}$)
03 Assert size($\vec{y}$) = n
04 $k \leftarrow y_1 \oplus \ldots \oplus y_n$
05 Return $k$
```

Figure 7: Construction for n out of n setting.

```
Proc setup($\vec{y}$)
00 Assert size($\vec{y}$) = n
01 $k \leftarrow_{\$} \mathcal{K}$
02 $\vec{\alpha} \leftarrow$ SecShare($k$, t, n)
03 $\vec{SV} \leftarrow \vec{\alpha} + \vec{y}$
04 Return $k, \vec{SV}$

Proc reconstruct($\vec{z}, \vec{SV}$)
05 Assert size($\vec{z}$) = t
06 $S = \emptyset$
07 For each $z_i \neq \perp$:
08     $\alpha_i \leftarrow \vec{SV}_i - z_i$
09     $\vec{\alpha}' \xleftarrow{\cup} \{\alpha_i\}$
10 $k \leftarrow$ SecCombine($\vec{\alpha}'$)
11 return $k$
```

Figure 8: Construction for t out of n setting.

user will locally run setup to acquire a vector of values, that can be stored alongside its ciphertexts at the storage server, where each entry is an OPRF output summed with a secret sharing of the user's key. This idea was used by Everspaugh *et al.* [ECS+15a] in the threshold version of their OPRF system. Later, the user can rederive the file encryption key by interacting with at least t servers. If reconstruction (or file decryption) fails, the user can retry with a different subset of servers. If F is verifiable then the user can identify if a server has not responded correctly, and omit that result from the reconstruct phase.

Conceptually this scheme is quite different to the n out of n scheme in Sec. 4.2. The file encryption key k is generated randomly by the user of the system, rather than as a function of the user password and the OPRF keys of the servers. This does not necessarily imply it has to be sampled on the device though. Indeed, we can bootstrap the procedure by first running an n′ out of n′ scheme for $t \leq n' \leq n$ with an initially trusted subset of the servers to generate the random key k, and for most applications it would be prudent to do so.

4.4 Security Proofs

We first provide the theorems and proofs for KIND-1 security and subsequently for KIND-2, reducing the security of our construction to the PRIV-1 and PRIV-2, respectively, security of the underlying OPRF and the PRNG security of the underlying OPRF. While the t out of n case is a generalization of the n out of n, we still provide a proof for the special n out of n case as it is more instructive, and the proof is easily adaptable to the generic t out of n case.

Theorem 4.1. *Let* PERKS *be an* n*-out-of-*n DKA *scheme built using OPRF* F *according to Fig.* 6 *and Fig.* 7*. For any adversary* $\mathcal{A}$ *against the* KIND-1 *security of* PERKS*, there exist adversaries* $\mathcal{B}$ *and* $\mathcal{C}$ *against the* PRIV-1 *and* PRNG *security of* F *respectively, such that*

$$\mathbf{Adv}^{\text{KIND-1}}_{\mathsf{PERKS}}(\mathcal{A}, n, n) \leq n \cdot \left(2 \cdot \mathbf{Adv}^{\text{PRIV-1}}_{\mathsf{F}}(\mathcal{B}) + \mathbf{Adv}^{\text{PRNG}}_{\mathsf{F}}(\mathcal{C}, 1) + \frac{q}{|\mathcal{D}|} \right).$$

Proof Intuition. We will show a reduction from the KIND-1 to KIND′-1, where the server that may not be corrupted is fixed at the start of the game. Subsequently we will bound the KIND′-1 advantage using a sequence of game hops, starting with the $b = 0$ side where a real key is returned to the adversary and ending with the $b = 1$ side (random key). To provide a reduction to PRNG security we need to embed the PRNG challenge in one of the servers, which means that we will not be able to answer Corrupt queries for that index. To do this, we pick the server in the KIND′-1 game that may not be corrupted. The reduction from KIND-1 to KIND′-1 invokes a loss of $\frac{1}{n}$. Then, for the majority of this proof we will calculate the advantage of an adversary attempting to distinguish in which game it is playing, to bound the advantage of the KIND′-1 game.

Proof. In game KIND′-1, the environment is identical to KIND-1 except that it picks a random index j out of all n servers at the start of the game and the adversary loses if it calls Corrupt on index j. We simulate KIND-1 by simply forwarding all oracle queries to KIND′-1. Given that the adversary may corrupt up to $(\mathrm{n}-1)$ servers in the course of its execution and the index j in KIND′-1 is picked uniformly at random independently of the adversary, the probability that server j will be corrupted is bounded by $1-\frac{1}{n}$. Thus, with probability at least $\frac{1}{n}$, game KIND′-1 will not abort and the simulation succeeds, as the games are identical in this case. As a result,

$$\mathbf{Adv}^{\text{KIND-1}}_{\mathsf{PERKS}}(\mathcal{A}) \leq \mathrm{n} \cdot \mathbf{Adv}^{\text{KIND}'\text{-1}}_{\mathsf{PERKS}}(\mathcal{A}).$$

We proceed to bound $\mathbf{Adv}^{\text{KIND}'\text{-1}}_{\mathsf{PERKS}}(\mathcal{A})$ using a sequence of games G_i, and define $\epsilon_i = \Pr\left[\mathsf{G}^{G_i}_{\mathsf{PERKS}}(\mathcal{A}) = 1\right]$. Game G_0 is the $b=0$ side, i.e. with the key returned in the Challenge query being the key computed in the protocol (if it exists) of the KIND′-1 game, and consequently $\epsilon_0 = \Pr\left[\mathsf{G}^{\text{KIND}'\text{-1}^0}_{\mathsf{PERKS}}(\mathcal{A}) = 1\right]$.

In game G_1, the environment is identical to G_0 except that for every user identity uid, the challenger will create two passwords: one will be used in Init, Acquire and Recover queries, and the other in Challenge and Reveal queries. Note that Reveal returns $\perp$ for any password that is not the selected password.

Intuitively, an adversary that can distinguish these games can infer information about the password or key from the $\mathrm{req}, \mathrm{rep}$ values it sees from interacting with Init, Acquire and Recover, so it notices when a different password has been used in the Reveal and Challenge queries. From such an adversary we build a reduction with similar advantage against PRIV-1 of the underlying OPRF F.

The reduction $\mathcal{B}$ is detailed in Fig. 9. $\mathcal{B}$ plays the PRIV-1^b game and simulates the KIND′-1^0 game (G_0) or its two-password version (G_1) to $\mathcal{A}$. Let b' be the output bit of $\mathcal{A}$, i.e. its indication of which game $G_{b'}$ that $\mathcal{A}$ believes it is playing. To create the simulation, $\mathcal{B}$ selects pw_0, pw_1 for each uid and generates OPRF key pairs for each of the n key servers. When $\mathcal{A}$ calls Init or Acquire for some uid, the reduction will look up the two passwords pw_0, pw_1 associated with that user identity and call its own Challenge(uid, pw_0, pw_1) oracle and receive $(\mathrm{req}_b, \mathrm{req}_{1-b})$. $\mathcal{B}$ then uses secret keys for each OPRF server to produce rep_i values for req_b. Moreover, $\mathcal{B}$ computes OPRF outputs y_i for pw_0 and the user key k, to be used for Reveal and Challenge queries. Importantly, we already want to remark here that $\mathcal{B}$ will simulate G_0 if it is playing the PRIV-1^0 game (because the req, rep and k are all consistent with pw_0) and $\mathcal{B}$ will simulate G_1 if it is playing the PRIV-1^1 game (because k is derived from pw_0 and req and rep are derived from pw_1).

Acquire queries are handled similarly to Init queries, with the difference being that the OPRF output values y_i are simply stored by $\mathcal{B}$ in array Y indexed by reconstruct counter r, instead of being used to compute the key immediately. For Recover queries, the input given by the adversary is $(r, \mathrm{C}, \vec{\mathrm{SV}})$, and recall that in the n-out-of-n construction the $\vec{\mathrm{SV}}$ are ignored and key reconstruction will fail if the chosen server set C is anything other than the full set of servers, i.e. $\mathrm{C} = (\mathrm{S}_1, \ldots, \mathrm{S}_\mathrm{n})$. This means the only interesting input is r, but the adversary has no control over the (deterministic) operations involved that will reconstruct the user key for password pw_0 for the uid corresponding with reconstruct counter r.

For queries to Reveal of the form (pw', uid), the reduction simply returns $\mathrm{K}[pw', uid]$. This will either be $\times$ if no value has been set or potentially user key k if $pw' = pw_0$ and $\mathrm{K}[pw_0, uid]$ has already been set. For Challenge(uid) queries, $\mathcal{B}$ checks if the uid has been queried before, and if not it will return k. To answer a Corrupt query, $\mathcal{B}$ needs to check if the query is allowed and **abort** otherwise, but $\mathcal{A}$ would lose anyway as this would be an illegal oracle query in both games.

As we remarked above, if $\mathcal{B}$ is playing PRIV-1^0, it will provide $\mathrm{req}, \mathrm{rep}$ and k values to $\mathcal{A}$ that are consistent with each other (and consistent with password pw_0), and thus this is a perfect simulation of G_0. If $\mathcal{B}$ is playing PRIV-1^1, then pw_0 governs Challenge and Reveal queries, while pw_1 governs Init and Reconstruct queries and thus this is a perfect simulation of G_1.

It is left to argue that $\mathcal{A}$'s success in distinguishing these games carries over to an advantage for $\mathcal{B}$. Intuitively, a win for $\mathcal{A}$ implies some way of linking $(\mathrm{req}, \vec{\mathrm{rep}})$ tuples to the user keys output by the protocol in the G_0 case, or noticing the absence of such a link in the G_1 case (to see this, consider $\mathrm{n}=1$ and a protocol where $k = pw$ and $(\mathrm{req}, \vec{\mathrm{rep}})$ information theoretically hide pw and k: an adversary has no way of distinguishing G_0 from G_1). This implies that $\mathcal{A}$ gains some information from its $(\mathrm{req}, \vec{\mathrm{rep}})$ values: if $b' = 0$ then $\mathcal{A}$ believes that its oracles are all running the same password, and thus $(\mathrm{req}_b, \vec{\mathrm{rep}})$ is linked to k, so $\mathcal{B}$ outputs 0; if $b' = 1$ then $\mathcal{A}$ thinks its oracles have been separated, and $\mathcal{B}$ outputs 1.

To conclude, any advantage for $\mathcal{A}$ directly corresponds to the advantage for the reduction $\mathcal{B}$:

$$\epsilon_0 - \epsilon_1 \leq \mathbf{Adv}_{\mathsf{F}}^{\text{PRIV-1}}(\mathcal{B}).$$

```
**Reduction $\mathcal{B}$ playing** PRIV-$1^b$
$\mathrm{K}[\cdot] \leftarrow \perp$; $\mathrm{PW}[\cdot] \leftarrow \perp$; $\mathrm{Y}[\cdot] \leftarrow \perp$
For $uid \in \mathcal{UID}$:
    $pw_0, pw_1 \leftarrow_\$ \mathcal{D}$
    $\mathrm{PW}[uid] \leftarrow pw_0, pw_1$
$\mathrm{CH} \leftarrow \emptyset$; $\mathrm{CO} \leftarrow \emptyset$
$r \leftarrow 0$
For $i \in [1 .. \mathrm{n}]$:
    $(\mathrm{sk}_i, \mathrm{pk}_i) \leftarrow \mathsf{F.KG}$
$b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
Return $b'$

**Oracle** Init$(uid, \vec{\mathrm{pk}})$
$pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
**call** Challenge$_{\mathcal{B}}(uid, pw_0, pw_1)$
**receive** $(\mathrm{req}_b, \mathrm{req}_{1-b})$
For $i \in [1 .. \mathrm{n}]$:
    $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
    $\mathrm{rep}_i \leftarrow \mathsf{F.BlindEv}(csk_i, uid, \mathrm{req}_b)$
    $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
Try: $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{setup}(\vec{\mathrm{y}})$
$\mathrm{K}[pw_0, uid] \leftarrow k$
Return $(\mathrm{req}_b, \vec{\mathrm{rep}}, \vec{\mathrm{SV}})$

**Oracle** Corrupt$(i)$
Require $i \neq j$
$\mathrm{CO} \overset{\cup}{\leftarrow} \{i\}$
Return $\mathrm{sk}_i$
```

```
**Oracle** Acquire$(uid, \vec{\mathrm{pk}})$
$r \overset{+}{\leftarrow} 1$
$pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
**call** Challenge$_{\mathcal{B}}(uid, pw_0, pw_1)$
**receive** $(\mathrm{req}_b, \mathrm{req}_{1-b})$
For $i \in [1 .. \mathrm{n}]$:
    $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
    $\mathrm{rep}_i \leftarrow \mathsf{F.BlindEv}(csk_i, uid, \mathrm{req}_b)$
    $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
$\mathrm{Y}[r] \leftarrow (\mathrm{y}_1, \ldots, \mathrm{y}_\mathrm{n})$
Return $(\mathrm{req}_b, \vec{\mathrm{rep}}, r)$

**Oracle** Recover$(r, \mathrm{C}, \vec{\mathrm{SV}})$
$\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
Try: $k \leftarrow \mathsf{DKA.recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
$\mathrm{K}[pw_0, uid] \leftarrow k$
Return

**Oracle** Challenge$_{\mathcal{A}}(uid)$
Require $uid \notin \mathrm{CH}$
$pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
$\mathrm{CH} \overset{\cup}{\leftarrow} \{uid\}$
$k \leftarrow \mathrm{K}[pw_0, uid]$
Return $k$

**Oracle** Reveal$(pw', uid)$
$k \leftarrow \mathrm{K}[pw', uid]$
Return $k$
```

Figure 9: Reduction $\mathcal{B}$ for the proof of Theorem 4.1 and Theorem 4.2. Procedures setup and reconstruct as in Fig. 7 for Thm. 4.1 and as in Fig. 8 for Thm. 4.2.

In game G_2, the environment is identical to G_1 except that the Reveal and Challenge oracles return a random element of the key space. For interactions with S_j, the reduction will replace the function $\mathsf{F.Ev}(\mathrm{sk}_j, \cdot, \cdot)$ by a random function of the same domain and range, where blinded evaluation queries are simulated. Recall S_j is the randomly picked server that the adversary may not corrupt. This invokes a reduction $\mathcal{C}$ to the PRNG security of OPRF F. We show that we can use an adversary $\mathcal{A}$ that distinguishes between G_1 and G_2 to win the PRNG game with the same advantage, i.e.:

$$\epsilon_1 - \epsilon_2 \leq \mathbf{Adv}_{\mathsf{F}}^{\text{PRNG}}(\mathcal{C}, 1).$$

Let $\mathcal{C}$ play the PRNG game and simulate the KIND′-1 game (more specifically, either G_1 or G_2) to $\mathcal{A}$. The reduction is detailed in Fig. 10.

The reduction $\mathcal{C}$ receives a public key for its own $\text{PRNG}^b(\mathcal{C}, 1)$ game, and then chooses two passwords for each user: pw_0 for Reveal and Challenge queries, and pw_1 for Init and Reconstruct queries. It is with the uncorrupted server S_j's interactions that $\mathcal{C}$ will embed its own queries. For any query to Init or Reconstruct, the reduction $\mathcal{C}$ needs to call its own Ev oracle with pw_0 to receive the session key share y_j that will be set for future Challenge and Reveal queries, and its own BlindEv oracle with pw_1 to acquire rep_j that the adversary $\mathcal{A}$ expects to receive. Note that the user key is only set for pw_0.

To answer a Corrupt query, $\mathcal{C}$ needs to check if the query is allowed and **abort** otherwise, but $\mathcal{A}$ would lose anyway as this would be an illegal oracle query in both games. For Reveal queries on (pw', uid), the reduction simply returns $\mathrm{K}[pw', uid]$. This will either be $\times$ if no value has been set or potentially k if $pw' = pw_0$ and $\mathrm{K}[pw_0, uid]$ has already been set. For Challenge(uid) queries, $\mathcal{C}$ first checks if the uid has been queried before, and if not it will return k. Eventually, $\mathcal{C}$ outputs to its own challenger whatever $\mathcal{A}$ outputs.

In the event that $\mathcal{C}$ is playing PRNG^0, the responses to its own queries will be the real F, and thus this perfectly simulates game G_1 for $\mathcal{A}$. If $\mathcal{C}$ is playing PRNG^1 then req and the rep values lie in the correct space but for some other randomly chosen function, and the key share for server S_j is an output of this random function, so the user key returned in Challenge is an output of a random function XORed with 'genuine' key shares, which is equivalent to choosing a random element of the key space. Thus this is a perfect simulation of G_2 for $\mathcal{A}$.

```
**Reduction** $\mathcal{C}$ **playing** $\mathrm{PRNG}^b(\mathcal{C}, 1)$
**receive** $\mathrm{pk}_j$
$\mathrm{K}[\cdot] \leftarrow \perp$
$\mathrm{PW}[\cdot] \leftarrow \perp$
For $uid \in \mathcal{UID}$:
    $pw_0, pw_1 \leftarrow_\$ \mathcal{D}$
    $\mathrm{PW}[uid] \leftarrow pw_0, pw_1$
$\mathrm{CH} \leftarrow \emptyset; \mathrm{CO} \leftarrow \emptyset$
For $i \in [1 \mathinner{..} \mathrm{n}] \setminus \{j\}$:
    $(\mathrm{sk}_i, \mathrm{pk}_i) \leftarrow \mathsf{F.KG}$
$b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
Return $b'$

**Oracle** Corrupt$(i)$
Require $i \neq j$
$\mathrm{CO} \overset{\cup}{\leftarrow} \{i\}$
Return $\mathrm{sk}_i$

**Oracle** Init$(uid, \vec{\mathrm{pk}})$
$pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
**call** $\mathrm{Ev}(uid, pw_0)$
**receive** y
$\mathrm{y}_j \leftarrow \mathrm{y}$
$(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
**call** $\mathrm{BlindEv}(uid, \mathrm{req})$
**receive** rep
$\mathrm{rep}_j \leftarrow \mathrm{rep}$
For $i \in [1 \mathinner{..} \mathrm{n}] \setminus \{j\}$:
    $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
    $\mathrm{rep}_i \leftarrow \mathsf{F.BlindEv}(csk_i, uid, \mathrm{req})$
    $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
Try: $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{setup}(\vec{\mathrm{y}})$
$\mathrm{K}[pw_0, uid] \leftarrow k$
Return $(\mathrm{req}, \vec{\mathrm{rep}}, \vec{\mathrm{SV}})$

**Oracle** Acquire$(uid, \vec{\mathrm{pk}})$
$r \overset{+}{\leftarrow} 1$
$pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
**call** $\mathrm{Ev}(uid, pw_0)$
**receive** y
$\mathrm{y}_i \leftarrow \mathrm{y}$
$(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
**call** $\mathrm{BlindEv}(uid, \mathrm{req})$
**receive** rep
$\mathrm{rep}_i \leftarrow \mathrm{rep}$
For $i \in [1 \mathinner{..} \mathrm{n}] \setminus \{j\}$:
    $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
    $\mathrm{rep}_i \leftarrow \mathsf{F.BlindEv}(csk_i, uid, \mathrm{req})$
    $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
$\mathrm{Y}[r] \leftarrow (\mathrm{y}_1, \ldots, \mathrm{y}_\mathrm{n})$
Return $(\mathrm{req}, \vec{\mathrm{rep}}, r)$

**Oracle** Recover$(r, \mathrm{C}, \vec{\mathrm{SV}})$
$\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
Try: $k \leftarrow \mathsf{DKA.recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
$\mathrm{K}[pw_0, uid] \leftarrow k$
Return

**Oracle** Reveal$(pw', uid)$
$k \leftarrow \mathrm{K}[pw', uid]$
Return $k$

**Oracle** Challenge$_\mathcal{A}(uid)$
Require $uid \notin \mathrm{CH}$
$pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
$\mathrm{CH} \overset{\cup}{\leftarrow} \{uid\}$
$k \leftarrow \mathrm{K}[pw_0, uid]$
Return $k$
```

Figure 10: Reduction $\mathcal{C}$ for the proof of Theorem 4.1. Procedures setup and reconstruct as in Fig. 7.

In game G_2 the Reveal and Challenge queries return a random element of the key space. Thus in G_3 we make a change to the Reveal oracle to use the key derived from pw_0 again. In all games up until this point the Reveal and Challenge oracles have been consistent with each other, but in G_3 they are not. Note that the adversary can only notice a difference between G_2 and G_3 if it queries Reveal on pw_0, as Reveal returns $\perp$ for all other passwords and the other oracles are identical. We remark that in both games Init and Reconstruct use pw_1 and Challenge simply samples a random key from the key space, so no information about pw_0 can be leaked from the oracle queries. Hence, the best any adversary can do is query the Reveal oracle for a randomly guessed password.

$$\epsilon_2 - \epsilon_3 \leq \frac{\mathrm{q}}{|\mathcal{D}|}.$$

In game G_4 we re-merge queries such that for a given user identity uid, queries to Init, Reconstruct and Reveal are all associated with a single password. In a very similar manner to the hop between G_0

and G_1, this invokes a PRIV-1 term. The reduction itself is almost identical to reduction $\mathcal{B}$ in Fig. 9, except that line 40 is replaced by selection of a random key from the key space.

$$\epsilon_3 - \epsilon_4 \le \mathbf{Adv}_{\mathsf{F}}^{\text{PRIV-1}}(\mathcal{B}).$$

Game G_4 is the $b = 1$ side, i.e. with the key returned in the Challenge query being a randomly chosen key, of the KIND′-1 game. Consequently $\epsilon_4 = \Pr\left[\mathsf{G}_{\mathsf{PERKS}}^{\text{KIND}'\text{-1}^1}(\mathcal{A})\right]$.

Collecting the terms results in the claimed bound, since

$$\mathbf{Adv}_{\mathsf{PERKS}}^{\text{KIND-1}}(\mathcal{A}) \le \mathrm{n} \cdot \mathbf{Adv}_{\mathsf{PERKS}}^{\text{KIND-1}'}(\mathcal{A}), \text{ and}$$

$$\begin{aligned}\mathbf{Adv}_{\mathsf{PERKS}}^{\text{KIND}'\text{-1}}(\mathcal{A}) &= \left|\Pr\left[\mathsf{G}_{\mathsf{PERKS}}^{\text{KIND}'\text{-1}^1}(\mathcal{A})\right] - \Pr\left[\mathsf{G}_{\mathsf{PERKS}}^{\text{KIND}'\text{-1}^0}(\mathcal{A})\right]\right| = |\epsilon_4 - \epsilon_0| \\ &= |\epsilon_0 - \epsilon_1 + \epsilon_1 - \epsilon_2 + \epsilon_2 - \epsilon_3 + \epsilon_3 - \epsilon_4| \\ &\le \mathbf{Adv}_{\mathsf{F}}^{\text{PRIV-1}}(\mathcal{B}) + \mathbf{Adv}_{\mathsf{F}}^{\text{PRNG}}(\mathcal{C}, 1) + \frac{\mathrm{q}}{|\mathcal{D}|} + \mathbf{Adv}_{\mathsf{F}}^{\text{PRIV-1}}(\mathcal{B}).\end{aligned}$$

□

Theorem 4.2. *Let* PERKS *be an* t*-out-of-*n DKA *scheme built using OPRF* F *according to Fig. 6 and Fig. 8 for* t *such that* $1 \le \mathrm{t} \le \mathrm{n}$*. For any adversary* $\mathcal{A}$ *against the* KIND-1 *security of* PERKS*, there exist adversaries* $\mathcal{B}$ *and* $\mathcal{C}$ *against the* PRIV-1 *and* PRNG *security of* F *respectively, such that*

$$\mathbf{Adv}_{\mathsf{PERKS}}^{\text{KIND-1}}(\mathcal{A}, \mathrm{t}, \mathrm{n}) \le \binom{\mathrm{n}}{\mathrm{t}-1} \cdot \left(2 \cdot \mathbf{Adv}_{\mathsf{F}}^{\text{PRIV-1}}(\mathcal{B}) + \mathbf{Adv}_{\mathsf{F}}^{\text{PRNG}}(\mathcal{C}, \mathrm{n}-\mathrm{t}+1) + \frac{\mathrm{q}}{|\mathcal{D}|}\right).$$

Proof Sketch. We remark Theorem 4.1 is the special case $\mathrm{t} = \mathrm{n}$ of this theorem and the game hops are very similar to that proof. For brevity we only provide the modifications to the proof here rather than duplicating the proof in its entirety. We also believe that only highlighting the steps where the proof needs to be generalized increases clarity.

For the first step, the success probability of the simulation now depends on t, as the reduction needs to select $\mathrm{n} - \mathrm{t} + 1$ uncorrupted servers. We denote this set of indices of uncorrupted servers with J. A lower bound for the success probability is provided by the number of ways to select $(\mathrm{t} - 1)$ servers out of $(\mathrm{t} - 1)$ servers divided by the number of ways to select $(\mathrm{t} - 1)$ servers out of n servers:

$$\frac{1}{\binom{\mathrm{n}}{\mathrm{t}-1}}.$$

Note that for $\mathrm{t} = \mathrm{n}$ we obtain the lower bound $\frac{1}{\mathrm{n}}$ from Theorem 4.1.

The reduction $\mathcal{B}$ in Fig. 9 from the indistinguishability between G_0 and G_1 to the PRIV-1 game is the same as in Theorem 4.1 with the trivial modification that it uses the t out of n setup and reconstruct procedures from Fig. 8 instead of the procedures from Fig. 7. We apply the same modification to the reduction $\mathcal{C}$ from the indistinguishability between G_1 and G_2 to the PRNG game. Moreover, the special case $i = j$, where j is the index of the uncorrupted server in reduction $\mathcal{C}$ is now generalized to $i \in J$, where J is the set of indices of uncorrupted servers. For completeness we provide the updated reduction in Fig. 11, but intuitively nothing novel happens in the reduction.

We need to argue replacing XOR in the setup and reconstruct procedures with a key sharing scheme also simulates G_2, i.e. the selected key is a random element from the key space. It is clear the adversary can have at most $(\mathrm{t} - 1)$ 'genuine' key shares because $(\mathrm{n} - \mathrm{t} + 1)$ servers return a random element. By the security of the secret sharing scheme, with $(\mathrm{t} - 1)$ key shares, any $k \in \mathcal{K}$ can still be reconstructed. Thus, if key share $s_{\mathrm{t}} \in \mathcal{K}$ is a random element of the key space, then so is the reconstructed key. It is clear this holds as s_{t} is the XOR of $\vec{\mathrm{SV}}_{\mathrm{t}}$ and y_{t}, where y_{t} is a random element of $\mathcal{K}$.

There is no modification to the hop from G_2 to G_3. The final hop from G_3 to G_4 is again the same as in in Theorem 4.1 with the trivial modification that it uses the t out of n setup and reconstruct procedures from Fig. 8. Collecting the terms yields the claimed result.

We now provide the theorems and proofs for KIND-2 security. For completeness we provide the modified reductions but as the modifications are trivial the descriptions will be brief. The theorems

```
**Reduction** $\mathcal{C}$ **playing** $\text{PRNG}^b(\mathcal{C}, \text{n} - \text{t} + 1)$
**receive** $\vec{\text{pk}}'$
$\text{K}[\cdot] \leftarrow \bot$; $\text{PW}[\cdot] \leftarrow \bot$
For $uid \in \mathcal{UID}$:
    $pw_0, pw_1 \leftarrow_\$ \mathcal{D}$
    $\text{PW}[uid] \leftarrow pw_0, pw_1$
$\text{CH} \leftarrow \emptyset; \text{CO} \leftarrow \emptyset$
For $i \in J$:
    $\text{pk}_i \leftarrow \text{pk}'_{\sigma(i)}$
For $i \in [1 \mathinner{..} \text{n}] \setminus J$:
    $(\text{sk}_i, \text{pk}_i) \leftarrow \mathsf{F.KG}$
$b' \leftarrow \mathcal{A}(\vec{\text{pk}})$
Return $b'$

**Oracle** $\text{Init}(uid, \vec{\text{pk}})$
$pw_0, pw_1 \leftarrow \text{PW}[uid]$
$(\text{req}, \text{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
For $i \in J$:
    **call** $\text{Ev}_{\sigma(i)}(uid, pw_0)$
    **receive** $\text{y}_i$
    **call** $\text{BlindEv}_{\sigma(i)}(uid, \text{req})$
    **receive** $\text{rep}_i$
For $i \in [1 \mathinner{..} \text{n}] \setminus J$:
    $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\text{sk}_i, uid)$
    $\text{rep}_i \leftarrow \mathsf{F.BlindEv}(csk_i, uid, \text{req})$
    $\text{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
Try: $(k, \vec{\text{SV}}) \leftarrow \text{setup}(\vec{\text{y}})$
$\text{K}[pw_0, uid] \leftarrow k$
Return $(\text{req}, \vec{\text{rep}}, \vec{\text{SV}})$

**Oracle** $\text{Reveal}(pw', uid)$
$k \leftarrow \text{K}[pw', uid]$
Return $k$
```

```
**Oracle** $\text{Acquire}(uid, \vec{\text{pk}})$
$r \xleftarrow{+} 1$
$pw_0, pw_1 \leftarrow \text{PW}[uid]$
$(\text{req}, \text{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
For $i \in J$:
    **call** $\text{Ev}_{\sigma(i)}(uid, pw_0)$
    **receive** $\text{y}_i$
    **call** $\text{BlindEv}_{\sigma(i)}(uid, \text{req})$
    **receive** $\text{rep}_i$
For $i \in [1 \mathinner{..} \text{n}] \setminus J$:
    $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\text{sk}_i, uid)$
    $\text{rep}_i \leftarrow \mathsf{F.BlindEv}(csk_i, uid, \text{req})$
    $\text{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
$\text{Y}[r] \leftarrow (\text{y}_1, \ldots, \text{y}_\text{n})$
Return $(\text{req}, \vec{\text{rep}}, r)$

**Oracle** $\text{Recover}(r, \text{C}, \vec{\text{SV}})$
$\vec{\text{y}} \leftarrow \text{Y}[r]$
Try: $k \leftarrow \mathsf{DKA.recover}(\vec{\text{y}}, \text{C}, \vec{\text{SV}})$
$\text{K}[pw_0, uid] \leftarrow k$
Return

**Oracle** $\text{Challenge}_\mathcal{A}(uid)$
Require $uid \notin \text{CH}$
$pw_0, pw_1 \leftarrow \text{PW}[uid]$
$\text{CH} \xleftarrow{\cup} \{uid\}$
$k \leftarrow \text{K}[pw_0, uid]$
Return $k$

**Oracle** $\text{Corrupt}(i)$
Require $i \notin J$
$\text{CO} \xleftarrow{\cup} \{i\}$
Return $\text{sk}_i$
```

Figure 11: Reduction $\mathcal{C}$ for the proof of Theorem 4.2. Procedures setup and reconstruct as in Fig. 8. J is a set of t indices that may not be corrupted. σ is a bijection of the uncorrupted indices in the KIND game to the indices in the underlying PRNG game.

bound the advantage by $\mathbf{Adv}^{\text{PRIV-2}}_{\mathsf{F}}(\mathcal{B})$ (instead of PRIV-1) and in the proof we only need to adapt the reductions to use the F.finalize procedure and the Finalize oracle in the PRIV-2 game to compute the OPRF output value (instead of using the F.ev procedure and the Ev oracle), since the response may now be maliciously formed.

Theorem 4.3. *Let* PERKS *be an* n*-out-of-*n DKA *scheme built using OPRF* F *according to Fig. 6 and Fig. 7. For any adversary* $\mathcal{A}$ *against the* KIND-2 *security of* PERKS*, there exist adversaries* $\mathcal{B}$ *and* $\mathcal{C}$ *against the* PRIV-2 *and* PRNG *security of* F *respectively, such that*

$$\mathbf{Adv}^{\text{KIND-2}}_{\mathsf{PERKS}}(\mathcal{A}, \text{n}, \text{n}) \leq \text{n} \cdot \left(2 \cdot \mathbf{Adv}^{\text{PRIV-2}}_{\mathsf{F}}(\mathcal{B}) + \mathbf{Adv}^{\text{PRNG}}_{\mathsf{F}}(\mathcal{C}) + \frac{\text{q}}{|\mathcal{D}|} \right).$$

Proof Sketch. The proof goes analogously to the proof of Theorem 4.1. We need to adapt the reductions as they cannot assume functionality and simply call the F.ev procedure or the Ev oracle, since the rep values may now be maliciously formed. Therefore, the reductions now use the F.finalize procedure and the Finalize oracle in the PRIV-2 game to compute the OPRF output value y (or receive $\bot$). The modifications are trivial but for completeness we provide the updated reductions here. The reduction $\mathcal{B}$ is detailed in Fig. 12 and reduction $\mathcal{C}$ is detailed in Fig. 13.

Theorem 4.4. *Let* PERKS *be an* t*-out-of-*n DKA *scheme built using OPRF* F *according to Fig. 6 and Fig. 8 for* t *such that* $1 \leq \text{t} \leq \text{n}$*. For any adversary* $\mathcal{A}$ *against the* KIND-2 *security of* PERKS*,* *there exist adversaries $\mathcal{B}$ and $\mathcal{C}$ against the* PRIV-2 *and* PRNG *security of* F *respectively, such that*

$$\mathbf{Adv}^{\text{KIND-2}}_{\mathsf{PERKS}}(\mathcal{A}, \mathrm{t}, \mathrm{n}) \leq \binom{\mathrm{n}}{\mathrm{t}-1} \cdot \left(2 \cdot \mathbf{Adv}^{\text{PRIV-2}}_{\mathsf{F}}(\mathcal{B}) + \mathbf{Adv}^{\text{PRNG}}_{\mathsf{F}}(\mathcal{C}, \mathrm{n}-\mathrm{t}+1) + \frac{\mathrm{q}}{|\mathcal{D}|}\right).$$

Proof Sketch. This proof effectively applies both the adaptations made in Theorem 4.2 and Theorem 4.3. The reduction $\mathcal{C}$ is detailed in Fig. 14. Reduction $\mathcal{B}$ is the same as in Theorem 4.3 with the trivial modification that it uses the t out of n setup and reconstruct procedures from Fig. 8 instead of the procedures from Fig. 7.

```
Reduction $\mathcal{B}$ playing PRIV-2$^b$
$K[\cdot] \leftarrow \bot$; $PW[\cdot] \leftarrow \bot$; $Y[\cdot] \leftarrow \bot$
For $uid \in \mathcal{UID}$:
    $pw_0, pw_1 \leftarrow_\$ \mathcal{D}$
    $PW[uid] \leftarrow pw_0, pw_1$
$CH \leftarrow \emptyset$; $CO \leftarrow \emptyset$
$r \leftarrow 0$
For $i \in [1..n]$:
    $(sk_i, pk_i) \leftarrow \mathsf{F.KG}$
$b' \leftarrow \mathcal{A}(\vec{pk})$
Return $b'$

Oracle Init$(uid, \vec{pk})$
$pw_0, pw_1 \leftarrow PW[uid]$
call Challenge$_\mathcal{B}(uid, pw_0, pw_1)$
receive $(j, req_b, req_{1-b})$
For $i \in [1..n]$:
    Await $rep^i_b \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, req_b)$
    Await $rep^i_{1-b} \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, req_{1-b})$
    $cpk_i \leftarrow \mathsf{DKA.Cli.PKG}(pk_i, uid)$
    call Finalize$_\mathcal{B}(j, cpk_i, rep^i_b, rep^i_{1-b})$
    receive $(y^i_0, y^i_1)$
Try: $(k, \vec{SV}) \leftarrow setup(\vec{y}_0)$
$K[pw_0, uid] \leftarrow k$
Return $\vec{SV}$

Oracle Challenge$_\mathcal{A}(uid)$
Require $uid \notin CH$
$pw_0, pw_1 \leftarrow PW[uid]$
$CH \xleftarrow{\cup} \{uid\}$
$k \leftarrow K[pw_0, uid]$
Return $k$

Oracle Acquire$(uid, \vec{pk})$
$r \xleftarrow{+} 1$
$pw_0, pw_1 \leftarrow PW[uid]$
call Challenge$_\mathcal{B}(uid, pw_0, pw_1)$
receive $(j, req_b, req_{1-b})$
For $i \in [1..n]$:
    Await $rep^i_b \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, req_b)$
    Await $rep^i_{1-b} \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, req_{1-b})$
    $cpk_i \leftarrow \mathsf{DKA.Cli.PKG}(pk_i, uid)$
    call Finalize$_\mathcal{B}(j, cpk_i, rep^i_b, rep^i_{1-b})$
    receive $(y^i_0, y^i_1)$
$Y[r] \leftarrow (y^1_0, \ldots, y^n_0)$
Return $r$

Oracle Recover$(r, C, \vec{SV})$
$\vec{y} \leftarrow Y[r]$
Try: $k \leftarrow \mathsf{DKA.recover}(\vec{y}, C, \vec{SV})$
$K[pw_0, uid] \leftarrow k$
Return

Oracle BlindEv$(i, uid, req)$
$rep_i \leftarrow \mathsf{DKA.server.op}(sk_i, uid, req)$
Return $rep_i$

Oracle Reveal$(pw', uid)$
$k \leftarrow K[pw', uid]$
Return $k$

Oracle Corrupt$(i)$
Require $i \neq j$
$CO \xleftarrow{\cup} \{i\}$
Return $sk_i$
```

Figure 12: Reduction $\mathcal{B}$ for the proof of Theorem 4.3 and Theorem 4.4. Procedures setup and reconstruct as in Fig. 7 for Thm. 4.3 and as in Fig. 8 for Thm. 4.4.

```
**Reduction $\mathcal{C}$ playing $\mathrm{PRNG}^b(\mathcal{C}, 1)$**
**receive** $\mathrm{pk}_j$
$\mathrm{K}[\cdot] \leftarrow \perp$; $\mathrm{PW}[\cdot] \leftarrow \perp$
For $uid \in \mathcal{UID}$:
    $pw_0, pw_1 \leftarrow_{\$} \mathcal{D}$
    $\mathrm{PW}[uid] \leftarrow pw_0, pw_1$
$\mathrm{CH} \leftarrow \emptyset$; $\mathrm{CO} \leftarrow \emptyset$
For $i \in [1 \mathinner{..} \mathrm{n}] \setminus \{j\}$:
    $(\mathrm{sk}_i, \mathrm{pk}_i) \leftarrow \mathsf{F.KG}$
$b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
Return $b'$

**Oracle** $\mathrm{Init}(uid, \vec{\mathrm{pk}})$
$pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
$(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
For $i \in [1 \mathinner{..} \mathrm{n}]$:
    Await $\mathrm{rep}_i \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, \mathrm{req})$
    If $i \neq j$:
        $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
        $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
    Else:
        **call** $\mathrm{Ev}(uid, pw_0)$
        **receive** $\mathrm{y}_i$
    $cpk_i \leftarrow \mathsf{DKA.Cli.PKG}(\mathrm{pk}_i, uid)$
    $\mathrm{y}'_i \leftarrow \mathsf{F.Finalize}(cpk_i, \mathrm{rep}_i, \mathrm{st})$
    If $\mathrm{y}'_i = \perp$:
        $\mathrm{y}_i \leftarrow \perp$
Try: $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{setup}(\vec{\mathrm{y}})$
$\mathrm{K}[pw_0, uid] \leftarrow k$
Return $\vec{\mathrm{SV}}$

**Oracle** $\mathrm{Challenge}_{\mathcal{A}}(uid)$
Require $uid \notin \mathrm{CH}$
$pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
$\mathrm{CH} \xleftarrow{\cup} \{uid\}$
$k \leftarrow \mathrm{K}[pw_0, uid]$
Return $k$

**Oracle** $\mathrm{Reveal}(pw', uid)$
$k \leftarrow \mathrm{K}[pw', uid]$
Return $k$

**Oracle** $\mathrm{Acquire}(uid, \vec{\mathrm{pk}})$
$r \xleftarrow{+} 1$
$pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
$(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
For $i \in [1 \mathinner{..} \mathrm{n}]$:
    Await $\mathrm{rep}_i \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, \mathrm{req})$
    If $i \neq j$:
        $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
        $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
    Else:
        **call** $\mathrm{Ev}(uid, pw_0)$
        **receive** $\mathrm{y}_i$
    $cpk_i \leftarrow \mathsf{DKA.Cli.PKG}(\mathrm{pk}_i, uid)$
    $\mathrm{y}'_i \leftarrow \mathsf{F.Finalize}(cpk_i, \mathrm{rep}_i, \mathrm{st})$
    If $\mathrm{y}'_i = \perp$:
        $\mathrm{y}_i \leftarrow \perp$
$\mathrm{Y}[r] \leftarrow (\mathrm{y}_1, \ldots, \mathrm{y}_\mathrm{n})$
Return $(\mathrm{req}, \vec{\mathrm{rep}}, r)$

**Oracle** $\mathrm{Recover}(r, \mathrm{C}, \vec{\mathrm{SV}})$
$\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
Try: $k \leftarrow \mathsf{DKA.recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
$\mathrm{K}[pw_0, uid] \leftarrow k$
Return

**Oracle** $\mathrm{BlindEv}(i, uid, \mathrm{req})$
If $i \neq j$:
    $\mathrm{rep} \leftarrow \mathsf{DKA.server.op}(\mathrm{sk}_i, uid, \mathrm{req})$
Else:
    **call** $\mathrm{BlindEv}(uid, \mathrm{req})$
    **receive** rep
Return rep

**Oracle** $\mathrm{Corrupt}(i)$
Require $i \neq j$
$\mathrm{CO} \xleftarrow{\cup} \{i\}$
Return $\mathrm{sk}_i$
```

Figure 13: Reduction $\mathcal{C}$ for the proof of Theorem 4.3. Procedures setup and reconstruct as in Fig. 7.

```
**Reduction** $\mathcal{C}$ **playing** $\mathrm{PRNG}^b(\mathcal{C}, \mathrm{n}-\mathrm{t}+1)$
**receive** $\vec{\mathrm{pk}}'$
$\mathrm{K}[\cdot] \leftarrow \perp$; $\mathrm{PW}[\cdot] \leftarrow \perp$
For $uid \in \mathcal{UID}$:
    $pw_0, pw_1 \leftarrow_\$ \mathcal{D}$
    $\mathrm{PW}[uid] \leftarrow pw_0, pw_1$
$\mathrm{CH} \leftarrow \emptyset$; $\mathrm{CO} \leftarrow \emptyset$
For $i \in J$:
    $\mathrm{pk}_i \leftarrow \mathrm{pk}'_{\sigma(i)}$
For $i \in [1\mathinner{..}\mathrm{n}] \setminus J$:
    $(\mathrm{sk}_i, \mathrm{pk}_i) \leftarrow \mathsf{F.KG}$
$b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
Return $b'$

**Oracle** $\mathrm{Init}(uid, \vec{\mathrm{pk}})$
$pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
$(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
For $i \in [1\mathinner{..}\mathrm{n}]$:
    Await $\mathrm{rep}_i \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, \mathrm{req})$
    If $i \in J$:
        **call** $\mathrm{Ev}_{\sigma(i)}(uid, pw_0)$
        **receive** $\mathrm{y}_i$
    If $i \notin J$:
        $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
        $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
    $cpk_i \leftarrow \mathsf{DKA.Cli.PKG}(\mathrm{pk}_i, uid)$
    $\mathrm{y}'_i \leftarrow \mathsf{F.Finalize}(cpk_i, \mathrm{rep}_i, \mathrm{st})$
    If $\mathrm{y}'_i = \perp$:
        $\mathrm{y}_i \leftarrow \perp$
Try: $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{setup}(\vec{\mathrm{y}})$
$\mathrm{K}[pw_0, uid] \leftarrow k$
Return $\vec{\mathrm{SV}}$

**Oracle** $\mathrm{Challenge}_{\mathcal{A}}(uid)$
Require $uid \notin \mathrm{CH}$
$pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
$\mathrm{CH} \overset{\cup}{\leftarrow} \{uid\}$
$k \leftarrow \mathrm{K}[pw_0, uid]$
Return $k$

**Oracle** $\mathrm{Acquire}(uid, \vec{\mathrm{pk}})$
$r \overset{+}{\leftarrow} 1$
$pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
$(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
For $i \in [1\mathinner{..}\mathrm{n}]$:
    Await $\mathrm{rep}_i \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, \mathrm{req})$
    If $i \in J$:
        **call** $\mathrm{Ev}_{\sigma(i)}(uid, pw_0)$
        **receive** $\mathrm{y}_i$
    If $i \notin J$:
        $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
        $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
    $cpk_i \leftarrow \mathsf{DKA.Cli.PKG}(\mathrm{pk}_i, uid)$
    $\mathrm{y}'_i \leftarrow \mathsf{F.Finalize}(cpk_i, \mathrm{rep}_i, \mathrm{st})$
    If $\mathrm{y}'_i = \perp$:
        $\mathrm{y}_i \leftarrow \perp$
$\mathrm{Y}[r] \leftarrow (\mathrm{y}_1, \ldots, \mathrm{y}_\mathrm{n})$
Return $(\mathrm{req}, \vec{\mathrm{rep}}, r)$

**Oracle** $\mathrm{Recover}(r, \mathrm{C}, \vec{\mathrm{SV}})$
$\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
Try: $k \leftarrow \mathsf{DKA}.\mathrm{recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
$\mathrm{K}[pw_0, uid] \leftarrow k$
Return

**Oracle** $\mathrm{BlindEv}(i, uid, \mathrm{req})$
If $i \in J$:
    **call** $\mathrm{BlindEv}_{\sigma(i)}(uid, \mathrm{req})$
    **receive** rep
Else:
    $\mathrm{rep} \leftarrow \mathsf{DKA.server.op}(\mathrm{sk}_i, uid, \mathrm{req})$
Return rep

**Oracle** $\mathrm{Reveal}(pw', uid)$
$k \leftarrow \mathrm{K}[pw', uid]$
Return $k$

**Oracle** $\mathrm{Corrupt}(i)$
Require $i \notin J$
$\mathrm{CO} \overset{\cup}{\leftarrow} \{i\}$
Return $\mathrm{sk}_i$
```

Figure 14: Reduction $\mathcal{C}$ for the proof of Theorem 4.4. Procedures setup and reconstruct as in Fig. 8. J is a set of t indices that may not be corrupted. σ is a bijection of the uncorrupted indices in the KIND game to the indices in the underlying PRNG game.

4.5 Use of Existing OPRFs in PERKS

As we have mentioned in Sec. 4, the DH-based VOPRFs in Fig. 2 allow the server to store one master key and compute private and public keys for users on the fly using *uid*: this operation is specified in Fig. 6. Remember that for non-verifiable OPRFs there is no public key and thus on-the-fly computation of per-user key material just needs to run a key derivation function from the server's (single) master key sk and *uid* to the same space as sk.

For non-DH VOPRFs, the DH group trick is not directly applicable, so either a similar trick using the structure of the public and secret keys needs to be found, or the server needs to store per-user key material. We regard finding such tricks in post-quantum VOPRFs as future work. The CSIDH-based scheme of Boneh *et al.* [BKW20] is not defined as a VOPRF, however this would appear to be a good candidate for a VOPRF that could fit with our DH trick.

For key rotation, the Pythia OPRF has no 'outer hash' (that destroys algebraic structure) and so is eligible for simple key rotation. Note that the aforementioned HashDH scheme can provide key rotation but only if the user stores inner hash values, but this is undesirable in our setting and modeling security for this case is not trivial.

This invokes a tradeoff: the Pythia OPRF provides key rotation at a computational cost (due to the pairing operation), while 2HashDH and 3HashSDHI are fast but without key rotation. As a result, the system designer needs to judge if the 'user initiated' key rotation methods in Sec. 5 are viable for the system's users, and if so 2HashDH or 3HashSDHI can be used.

Note that each of the three OPRFs in Fig. 2 are proven secure in different models, and our theorems relate to the security games of the 3HashSDHI scheme. Thus it remains to formally prove that the other two schemes do in fact meet POPRIV-x and PRNG security, or by showing that the proven security properties of the other schemes—VOPRF UC functionality for 2HashDH, and one-more unpredictability and one-more PRF for Pythia—are at least as strong as POPRIV-x and PRNG.

5 Using PERKS as a Storage System

We now explain why our approach is well-suited to derivation of a backup key for outsourced storage systems, and particularly for instant messaging. Then, we describe how our construction can be used to build a feature rich file system for cloud storage, incorporating recent work analyzing security of symmetric encryption schemes where a user encrypts 'to themself', deduplication, and efficient key rotation.

Instant messaging apps are generally free to download and use, and users are often unwilling to pay for additional features. This leaves very little room for maneuver when designing a secure backup service: users must use an internal solution like WhatsApp's (see Sec. 1.4), where the protocol is potentially strong but not open source. A service such as the one we propose needs to be extremely efficient in terms of bandwidth and storage to possibly be offered as a free service: this is why we aim to only use the most efficient OPRFs, and enforce minimal user and server storage. In particular, we envision OPRF services with multiple other roles in addition to PERKS, hence our system does not require the OPRF servers to be given a particular (share of a) key, as is done in many prior works [JKKX17, JKR19, BFH+20].

The constructions defined in Sec. 4 allow a user to derive a single symmetric key from a password. It remains to select a symmetric encryption primitive for encryption, a decision that is informed by the desired functionality and security properties.

Note that in the case of long-term encrypted backup, if a user's device is compromised and they wish to change their encryption key, they may still wish to recover messages stored under the old key (i.e. even if they believe that an adversary is already in possession of those messages). From this perspective, the user may wish to recover their messages after they have already chosen a new password for use with new messages, creating an overlap in the epochs of the system: this is a departure from the regular theoretical approach to key rotation via updatable encryption and we discuss this further below.

Encrypt-to-Self. Pijnenburg and Poettering [PP20] recently demonstrated that integrity protection can still be obtained in the event of user key corruption: if the user stores short file (ciphertext) hashes then even if the user knows that their key is corrupted they can check ciphertext integrity when downloading files and discard any where the hash does not match a local entry. In the same paper, the authors demonstrated a method to compute these hashes during encryption, to avoid making two passes over plaintext data.

Deduplication. If the user expects to upload some files many times, for example by backing up an entire disk periodically, and wants to avoid storing multiple copies of files then they can employ deduplication techniques such as convergent encryption [SGLM08, BKR13]. File key derivation for a file F could be for example $k_F \leftarrow \mathsf{H}(k||F)$ for some cryptographic hash function H.

Key Rotation. If the user wishes to rotate their file encryption key in PERKS then there are three possibilities:

1. Use an OPRF service that has automated key rotation, e.g. by using the Pythia OPRF [ECS+15a]. Note that for the n out of n construction, just one OPRF server updating its sk_i value results in a change in file encryption key. If this is used, then the server will provide 'tokens' that work similarly to updatable encryption (UE) update tokens: unblinded values provided under the old key can be efficiently modified to unblinded values under the new key, without the need to call the OPRF service under all the old inputs.
2. Use a different password. This will result in new OPRF output values for all OPRF servers. (Note that another credential modification technique is possible via an OPRF service that supports tweaks, i.e. different *uid* input values for the same user: this will result in different OPRF output values for the OPRF servers offering this.)
3. (t out of n construction only) Choose a new key k, essentially running setup again. This results in a new key share vector $\vec{\alpha}$ but the y_i values are unchanged so the user needs to publish a new public vector $\vec{\mathrm{SV}}$.

In all of these cases, the user can avoid downloading, decrypting, reencrypting and reuploading all of their files every time they update their file encryption key by utilizing updatable encryption [BLMR13, EPRS17, LT18, KLR19, BDGJ20, BEKS20], where the user can send a short update token to the ciphertext storage server (CSP) with which the underlying key for the ciphertexts can be rotated efficiently, without leaking information to the CSP. However the challenge is providing availability of key material in consecutive epochs. In the efficient (ciphertext-independent) UE schemes just mentioned, the update token calculation requires knowledge of an old key and a new key at the beginning of the new epoch. For user-initiated actions (items 2 and 3) this is trivial: the user runs the protocol to get their old key, then runs the protocol again using their new inputs, calculates the token, sends that to the file storage server and then deletes both keys locally. In the OPRF server key rotation setting (item 1) care is required: if a new epoch begins while the user does not have a local copy of the file encryption key available then the user would be locked out of access to their ciphertexts. To solve this issue the OPRF services could make a transition period available to users, where access is given to the OPRF functionality for the old and the new OPRF keys.

6 Discussion

We now discuss some of the design choices made in this work, the relationship with existing literature and some suggestions for future research.

The security definitions presented in this work are predominantly game-based rather than simulation-based, with the exception of the PRNG that is inherited from the work of Tyagi *et al.* [TCR+22] and adapted to our multi-server setting. Many papers in the literature present OPRF security in the universal composability framework [Can01] which reflects the fact that OPRFs are regularly composed with other primitives and are used in highly concurrent scenarios. We avoided using the UC model for our formalization of KIND-x due to the inherent requirement in the UC framework for the parties to agree on a session identifier sid before an instance of the protocol begins. It is of course trivial to agree on sid either via an additional round of communication or using counters stored at each party, however both of these options are extremely undesirable in the scenario that we consider. We consider it valuable future work to define a UC functionality that captures key acquisition with the same properties as our KIND-x games, with the additional benefit of composability.

At USENIX '15, Everspaugh *et al.* [ECS+15a] (ECSJR) presented the first definition of partially-oblivious OPRFs (POPRFs) and a candidate construction using pairings (more details in Sec. 2.5). Their motivating setting was a user engaging in password-based authentication with some web server, where instead of the web server storing password hashes, it would instead interact with 'a Pythia service' that hardened the user's password using a POPRF and a random per-user value as the OPRF public input t. In their approach, if an adversary gains access to the OPRF key then a pre-computed password database is of no use since the web server's stored values are functions of the key, the password and the random 'salt' t. This means that the Pythia service can detect online attacks and use rate limiting without learning the user password, and the web server never needs to learn the OPRF key. As we have stated, our threshold construction is essentially using the same idea as theirs, and the pairing-based POPRF presented by ECSJR is a candidate for use within PERKS. Our approach is for a different problem, namely a user interacting with multiple OPRF servers, and our security goals are therefore quite different. ECSJR used one-more unpredictability and one-more pseudorandomness as properties of the POPRF but do not consider any formal security properties for the use cases that motivated their paper, whereas in our case we needed to create KIND-x as a security definition that captures the indistinguishability of the keys used by the user in the eventual application. An interesting future research topic would be to categorize the security definitions for (P)OPRFs, to ascertain the 'correct' expectations of security. To provide a brief summary of the difficulties in comparing notions, Tyagi *et al.* explicitly referred to the ECSJR one-more pseudorandomness property as "non-standard" and advertise their own notions as avoiding the need for the ECSJR approach, while ECSJR provide a very brief sketch (in the ePrint version [ECS+15b] only) of the difficulty in proving their own POPRF UC secure (there is no outer hash because they seek key rotation, and this makes a UC proof nigh on impossible because the simulator can never see the necessary RO queries), followed by an unproven Claim that adding an extra outer hash gives a UC-secure VOPRF.

Our results in the threshold setting, namely Theorems 4.2 and 4.4, are not tight and lose a factor of $\binom{n}{t-1}$. It would of course be desirable to avoid this, using a proof technique that does not involve guessing the set of uncorrupted servers.

Finally, we remark that our approach defines an OPRF interaction as being two messages, namely a blinded representation of the password from the user to the server and then an application of the OPRF secret key to this blinded value in response. It is known how to construct (V)OPRFs with post-quantum security in this (round-optimal) manner [ADDS21] (see Sec. 2.4 for more details on existing P- and V-OPRF constructions), however it is still unclear if efficient constructions are feasible, and such constructions may not follow the two-message syntax that we use. In the event that relatively efficient verifiable OPRFs with post-quantum security can be constructed using some other syntax then firstly this would be a significant breakthrough, and further it would be necessary to adapt our formalism.

References

[ACNP16] Michel Abdalla, Mario Cornejo, Anca Nitulescu, and David Pointcheval. Robust password-protected secret sharing. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *ESORICS 2016, Part II*, volume 9879 of *LNCS*, pages 61–79. Springer, Heidelberg, September 2016.

[ADDS21] Martin R. Albrecht, Alex Davidson, Amit Deo, and Nigel P. Smart. Round-optimal verifiable oblivious pseudorandom functions from ideal lattices. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 261–289. Springer, Heidelberg, May 2021.

[BCP04] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. New security results on encrypted key exchange. In Feng Bao, Robert Deng, and Jianying Zhou, editors, *PKC 2004*, volume 2947 of *LNCS*, pages 145–158. Springer, Heidelberg, March 2004.

[BDGJ20] Colin Boyd, Gareth T. Davies, Kristian Gjøsteen, and Yao Jiang. Fast and secure updatable encryption. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 464–493. Springer, Heidelberg, August 2020.

[BEKS20] Dan Boneh, Saba Eskandarian, Sam Kim, and Maurice Shih. Improving speed and security in updatable encryption schemes. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 559–589. Springer, Heidelberg, December 2020.

[BEL$^+$20] Julian Brost, Christoph Egger, Russell W. F. Lai, Fritz Schmid, Dominique Schröder, and Markus Zoppelt. Threshold password-hardened encryption services. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 409–424. ACM Press, November 2020.

[BFH$^+$20] Carsten Baum, Tore Kasper Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. PESTO: proactively secure distributed single sign-on, or how to trust a hacked server. In Lujo Bauer and Frank Stajano, editors, *IEEE EuroS&P 2020*, pages 587–606. IEEE, 2020.

[BJSL11] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 433–444. ACM Press, October 2011.

[BKLW22] Daniel Bourdrez, Dr. Hugo Krawczyk, Kevin Lewi, and Christopher A. Wood. The OPAQUE Asymmetric PAKE Protocol. Internet-Draft draft-irtf-cfrg-opaque-08, Internet Engineering Task Force, March 2022. Work in Progress.

[BKM$^+$21] Andrea Basso, Péter Kutas, Simon-Philipp Merz, Christophe Petit, and Antonio Sanso. Cryptanalysis of an oblivious PRF from supersingular isogenies. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part I*, volume 13090 of *LNCS*, pages 160–184. Springer, Heidelberg, December 2021.

[BKR13] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 296–312. Springer, Heidelberg, May 2013.

[BKW20] Dan Boneh, Dmitry Kogan, and Katharine Woo. Oblivious pseudorandom functions from isogenies. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 520–550. Springer, Heidelberg, December 2020.

[BLMR13] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 410–428. Springer, Heidelberg, August 2013.

[BP13] Fabrice Benhamouda and David Pointcheval. Verifier-based password-authenticated key exchange: New models and constructions. Cryptology ePrint Archive, Report 2013/833, 2013. https://eprint.iacr.org/2013/833.

[BP14] Abhishek Banerjee and Chris Peikert. New and improved key-homomorphic pseudorandom functions. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 353–370. Springer, Heidelberg, August 2014.

[BS20] Dan Boneh and Victor Shoup. A graduate course in applied cryptography, 2020.

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

[CHL22] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. SoK: Oblivious pseudorandom functions. In David Evans and Carmela Troncoso, editors, *IEEE EuroS&P 2022*. IEEE, 2022.

[CLLN14] Jan Camenisch, Anja Lehmann, Anna Lysyanskaya, and Gregory Neven. Memento: How to reconstruct your secrets from a single password in a hostile environment. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 256–275. Springer, Heidelberg, August 2014.

[CLM$^+$18] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. CSIDH: An efficient post-quantum commutative group action. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 395–427. Springer, Heidelberg, December 2018.

[CLN12] Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. Practical yet universally composable two-server password-authenticated secret sharing. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 525–536. ACM Press, October 2012.

[DFHSW22] Alex Davidson, Armando Faz-Hernández, Nick Sullivan, and Christopher A. Wood. Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups. Internet-Draft draft-irtf-cfrg-voprf-09, Internet Engineering Task Force, February 2022. Work in Progress.

[DGS$^+$18] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. Privacy pass: Bypassing internet challenges anonymously. *PoPETs*, 2018(3):164–180, July 2018.

[DHL22] Poulami Das, Julia Hesse, and Anja Lehmann. DPaSE: Distributed password-authenticated symmetric-key encryption, or how to get many keys from one password. In Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIACCS 22*, pages 682–696. ACM Press, May / June 2022.

[DvMN08] Rafael Dowsley, Jeroen van de Graaf, Jörn Müller-Quade, and Anderson C. A. Nascimento. Oblivious transfer based on the McEliece assumptions. In Reihaneh Safavi-Naini, editor, *ICITS 08*, volume 5155 of *LNCS*, pages 107–117. Springer, Heidelberg, August 2008.

[ECS$^+$15a] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 547–562. USENIX Association, August 2015.

[ECS$^+$15b] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The Pythia PRF service. Cryptology ePrint Archive, Report 2015/644, 2015. https://eprint.iacr.org/2015/644.

[EPRS17] Adam Everspaugh, Kenneth G. Paterson, Thomas Ristenpart, and Samuel Scott. Key rotation for authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 98–129. Springer, Heidelberg, August 2017.

[FIPR05] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005.

[HIJ$^+$21] Sharon Huang, Subodh Iyengar, Sundar Jeyaraman, Shiv Kushwah, Chen-Kuei Lee, Zutian Luo, Payman Mohassel, Ananth Raghunathan, Shaahid Shaikh, Yen-Chieh Sung, and Albert Zhang. DIT: Deidentified authenticated telemetry at scale. Blog post, Meta, April 2021.

[JD11] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In Bo-Yin Yang, editor, *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*, pages 19–34. Springer, Heidelberg, November / December 2011.

[JKK14] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253. Springer, Heidelberg, December 2014.

[JKKX16] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In Michael Backes, editor, *IEEE European Symposium on Security and Privacy, EuroS&P 2016*, pages 276–291. IEEE, 2016.

[JKKX17] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 39–58. Springer, Heidelberg, July 2017.

[JKR18] Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch. Threshold partially-oblivious PRFs with applications to key management. Cryptology ePrint Archive, Report 2018/733, 2018. https://eprint.iacr.org/2018/733.

[JKR19] Stanislaw Jarecki, Hugo Krawczyk, and Jason K. Resch. Updatable oblivious key management for storage systems. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 379–393. ACM Press, November 2019.

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

[JL09] Stanislaw Jarecki and Xiaomin Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 577–594. Springer, Heidelberg, March 2009.

[KBR13] Sriram Keelveedhi, Mihir Bellare, and Thomas Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In Samuel T. King, editor, *USENIX Security 2013*, pages 179–194. USENIX Association, August 2013.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 818–829. ACM Press, October 2016.

[KLR19] Michael Klooß, Anja Lehmann, and Andy Rupp. (R)CCA secure updatable encryption with integrity protection. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 68–99. Springer, Heidelberg, May 2019.

[Leh19] Anja Lehmann. ScrambleDB: Oblivious (chameleon) pseudonymization-as-a-service. *PoPETs*, 2019(3):289–309, July 2019.

[LER$^+$18] Russell W. F. Lai, Christoph Egger, Manuel Reinert, Sherman S. M. Chow, Matteo Maffei, and Dominique Schröder. Simple password-hardened encryption services. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 1405–1421. USENIX Association, August 2018.

[LPA$^+$19] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1387–1403. ACM Press, November 2019.

[LT18] Anja Lehmann and Björn Tackmann. Updatable encryption with post-compromise security. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 685–716. Springer, Heidelberg, April / May 2018.

[NG21] NCC-Group. End-to-end encrypted backups security assessment: Whatsapp (version 1.2). https://research.nccgroup.com/wp-content/uploads/2021/10/NCC_Group_WhatsApp_E001000M_Report_2021-10-27_v1.2.pdf, 27th October 2021.

[PP20] Jeroen Pijnenburg and Bertram Poettering. Encrypt-to-self: Securely outsourcing storage. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020, Part I*, volume 12308 of *LNCS*, pages 635–654. Springer, Heidelberg, September 2020.

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 554–571. Springer, Heidelberg, August 2008.

[SGLM08] Mark W. Storer, Kevin M. Greenan, Darrell D. E. Long, and Ethan L. Miller. Secure data deduplication. In Yongdae Kim and William Yurcik, editors, *StorageSS 2008*, pages 1–10. ACM, 2008.

[Sha79] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

[SHB21] István András Seres, Máté Horváth, and Péter Burcsi. The legendre pseudorandom function as a multivariate quadratic cryptosystem: Security and applications. Cryptology ePrint Archive, Report 2021/182, 2021. https://eprint.iacr.org/2021/182.

[SKFB21] Nick Sullivan, Dr. Hugo Krawczyk, Owen Friel, and Richard Barnes. OPAQUE with TLS 1.3. Internet-Draft draft-sullivan-tls-opaque-01, Internet Engineering Task Force, February 2021. Work in Progress.

[TCR+22] Nirvan Tyagi, Sofía Celi, Thomas Ristenpart, Nick Sullivan, Stefano Tessaro, and Christopher A. Wood. A fast and simple partially oblivious PRF, with applications. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 674–705. Springer, Heidelberg, May / June 2022.

[TPY+19] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019*, pages 1556–1571. USENIX Association, August 2019.

[Wha21] WhatsApp. Security of end-to-end encrypted backups. https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf, 10th September 2021.

A OPRF Definition Relations

A.1 Comparison between Mechanics of PRIV-1 and POPRIV-1

We now summarize the differences between the POPRIV-1 game of Tyagi *et al.* [TCR+22] given in our notation in Fig. 15 and our PRIV-1 notion given in Fig. 3 (recall that our PRIV-2 game is identical to the POPRIV-2 game).

```
Game POPRIV-1^b(A)
00 $(\mathrm{pk},\mathrm{sk}) \leftarrow_\$ \mathsf{F.KG}$
01 $b' \leftarrow \mathcal{A}(\mathrm{pk},\mathrm{sk})$
02 Stop with $b'$

Oracle TRANS(uid, x_0, x_1)
03 $(\mathrm{req}_0,\mathrm{st}_0) \leftarrow \mathsf{F.Req}(\mathrm{t},\mathrm{x}_0)$
04 $(\mathrm{req}_1,\mathrm{st}_1) \leftarrow \mathsf{F.Req}(\mathrm{t},\mathrm{x}_1)$
05 $\mathrm{rep}_0 \leftarrow \mathsf{F.BlindEv}(\mathrm{sk},\mathrm{t},\mathrm{req}_0)$
06 $\mathrm{rep}_1 \leftarrow \mathsf{F.BlindEv}(\mathrm{sk},\mathrm{t},\mathrm{req}_1)$
07 $\mathrm{y}_0 \leftarrow \mathsf{F.Finalize}(\mathrm{pk},\mathrm{rep}_0,\mathrm{st}_0)$
08 $\mathrm{y}_1 \leftarrow \mathsf{F.Finalize}(\mathrm{pk},\mathrm{rep}_1,\mathrm{st}_1)$
09 $\tau \leftarrow (\mathrm{req}_b,\mathrm{rep}_b,\mathrm{y}_0)$
10 $\tau \leftarrow (\mathrm{req}_{1-b},\mathrm{rep}_{1-b},\mathrm{y}_1)$
11 Return $(\tau,\tau')$
```

Figure 15: POPRIV-1 security game [TCR+22].

In the POPRIV-1 game the adversary receives an OPRF key pair and access to a transcript oracle TRANS that runs the server side of the OPRF functionality using inputs $(uid, \mathrm{x}_0, \mathrm{x}_1)$ given by the adversary, and returns the request and response values in random order. In our PRIV-1 game the adversary is not given a key pair but has access to a challenge oracle Challenge with the same inputs $(uid, \mathrm{x}_0, \mathrm{x}_1)$, but only receives $(\mathrm{req}_b, \mathrm{req}_{1-b})$. Our observation is that the adversary can act as the server, i.e. running the deterministic process $\mathsf{F.BlindEv}$, for arbitrary key pairs. Recall that we give public key pk as an input to $\mathsf{F.Finalize}$ rather than embedding it in the state value st as done by [TCR+22].

A.2 PRIV-x and POPRIV-x are Equivalent

We now show the equivalence of our multi-server PRIV-x games and the single-server POPRIV-x games introduced by Tyagi *et al.* [TCR+22]. As stated earlier, our PRIV-2 game is identical to the POPRIV-2 game of Tyagi *et al.*, and so we focus on showing PRIV-1 ⇔ POPRIV-1.

Theorem A.1. *Let* F *be an oblivious pseudorandom function. For any adversary* $\mathcal{A}$ *against the* PRIV-1 *security of* F*, there exists an adversary* $\mathcal{B}$ *against the* POPRIV-1 *security of* F*, such that*

$$\mathbf{Adv}_{\mathsf{F}}^{\text{PRIV-1}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathsf{F}}^{\text{POPRIV-1}}(\mathcal{B}).$$

Proof. The direct reduction is detailed in Fig. 16. $\mathcal{B}$ runs $\mathcal{A}$, and needs to respond to $\mathcal{A}$'s calls to Challenge. Note that $\mathcal{A}$'s calls to Challenge give $(uid, \mathrm{x}_0, \mathrm{x}_1)$ as input, and $\mathcal{A}$ expects $(\mathrm{req}_b, \mathrm{req}_{1-b})$ in response, when it is playing PRIV-1^b. $\mathcal{B}$'s own oracle $\mathrm{TRANS}_{\mathcal{B}}$ provides a more detailed response, and so $\mathcal{B}$ simply takes the $\mathrm{req}_b, \mathrm{req}_{1-b}$ that it receives and forwards this to $\mathcal{A}$.

Let b be the challenge bit in the experiment that $\mathcal{B}$ is playing, and let b' be the bit that is output by $\mathcal{A}$. $\mathcal{B}$ receives $(\mathrm{req}_b, \mathrm{rep}_b, \mathrm{y}_0, \mathrm{req}_{1-b}, \mathrm{rep}_{1-b}, \mathrm{y}_1)$ from its own call to $\mathrm{TRANS}_{\mathcal{B}}$, and thus providing $(\mathrm{req}_b, \mathrm{req}_{1-b})$ to $\mathcal{A}$ simulates $\mathcal{A}$'s expected environment.

$\mathcal{B}$ perfectly simulates PRIV-1^b for $\mathcal{A}$, since the secret key vector is correctly distributed and the responses that $\mathcal{A}$ receives to its oracles calls are exactly as it would expect. The advantage of $\mathcal{A}$ directly corresponds to the advantage of $\mathcal{B}$. This concludes the proof.

```
Reduction $\mathcal{B}$ playing POPRIV-$1^b$
00 receive pk, sk
01 $b' \leftarrow \mathcal{A}()$
02 Return $b'$

Oracle $\mathrm{Challenge}_{\mathcal{A}}(uid, \mathrm{x}_0, \mathrm{x}_1)$
03 call $\mathrm{TRANS}_{\mathcal{B}}(uid, \mathrm{x}_0, \mathrm{x}_1)$
04 receive $(\mathrm{req}_b, \mathrm{rep}_b, \mathrm{y}_0, \mathrm{req}_{1-b}, \mathrm{rep}_{1-b}, \mathrm{y}_1)$
05 Return $(\mathrm{req}_b, \mathrm{req}_{1-b})$
```

Figure 16: Reduction $\mathcal{B}$ for the proof of Thm. A.1.

□

Theorem A.2. *Let* F *be an oblivious pseudorandom function. For any adversary* $\mathcal{A}$ *against the* POPRIV-1 *security of* F*, there exists an adversary* $\mathcal{B}$ *against the* PRIV-1 *security of* F*, such that*

$$\mathbf{Adv}_{\mathsf{F}}^{\text{POPRIV-1}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathsf{F}}^{\text{PRIV-1}}(\mathcal{B}).$$

Proof. The reduction is detailed in Fig. 17. In order to provide a sufficient response to $\mathcal{A}$, the reduction $\mathcal{B}$ must use the values $(\mathrm{req}_b, \mathrm{req}_{1-b})$ that it receives from $\mathrm{TRANS}_{\mathcal{A}}$ and perform $\mathsf{F.BlindEv}$ on them to acquire $(\mathrm{rep}_b, \mathrm{rep}_{1-b})$. Producing y_0 and y_1 is straightforward, since $\mathcal{B}$ can simply compute the OPRF evaluation with sk_j and the input values x_0 and x_1. $\mathcal{B}$ combines the values into output $(\mathrm{req}_b, \mathrm{rep}_b, \mathrm{y}_0, \mathrm{req}_{1-b}, \mathrm{rep}_{1-b}, \mathrm{y}_1)$ and returns it to $\mathcal{A}$. The reduction perfectly simulates the POPRIV-1^b environment for $\mathcal{A}$. This concludes the proof. □

```
Reduction $\mathcal{B}$ playing PRIV-$1^b$
00 $(\mathrm{pk}, \mathrm{sk}) \leftarrow \mathsf{F.KG}$
01 $b' \leftarrow \mathcal{A}(\mathrm{sk})$
02 Return $b'$

Oracle $\mathrm{TRANS}_{\mathcal{A}}(uid, \mathrm{x}_0, \mathrm{x}_1)$
03 call $\mathrm{Challenge}_{\mathcal{B}}(uid, \mathrm{x}_0, \mathrm{x}_1)$
04 receive $(\mathrm{req}_b, \mathrm{req}_{1-b})$
05 $\mathrm{rep}_b \leftarrow \mathsf{F.BlindEv}(\mathrm{sk}, uid, \mathrm{req}_b)$
06 $\mathrm{rep}_{1-b} \leftarrow \mathsf{F.BlindEv}(\mathrm{sk}, uid, \mathrm{req}_{1-b})$
07 $\mathrm{y}_0 \leftarrow \mathsf{F.Ev}(\mathrm{sk}, uid, \mathrm{x}_0)$
08 $\mathrm{y}_1 \leftarrow \mathsf{F.Ev}(\mathrm{sk}, uid, \mathrm{x}_1)$
09 Return $(\mathrm{req}_b, \mathrm{rep}_b, \mathrm{y}_0, \mathrm{req}_{1-b}, \mathrm{rep}_{1-b}, \mathrm{y}_1)$
```

Figure 17: Reduction $\mathcal{B}$ for the proof of Thm. A.2.