DPaSE: Distributed Password-Authenticated Symmetric Encryption*

Poulami Das[1], Julia Hesse[2], and Anja Lehmann[3]

[1]Technische Universität Darmstadt, Germany, poulami.das@tu-darmstadt.de
[2]IBM Research – Zurich, Switzerland, jhs@zurich.ibm.com
[3]Hasso-Plattner-Institute, University of Potsdam, Germany anja.lehmann@hpi.de

Abstract

Cloud storage is becoming increasingly popular among end users that outsource their personal data to services such as Dropbox or Google Drive. For security, uploaded data should ideally be encrypted under a key that is controlled and only known by the user. Current solutions that support user-centric encryption either require the user to manage strong cryptographic keys, or derive keys from weak passwords. While the former has massive usability issues and requires secure storage by the user, the latter approach is more convenient but offers only little security. The recent concept of password-authenticated secret-sharing (PASS) enables users to securely derive strong keys from weak passwords by leveraging a distributed server setup, and has been considered a promising step towards secure and usable encryption. However, using PASS for encryption is not as suitable as originally thought: it only considers the (re)construction of a *single*, static key – whereas practical encryption will require the management of *many*, object-specific keys. Using a dedicated PASS instance for every key makes the solution vulnerable against online attacks, inherently leaks access patterns to the servers and poses the risk of permanent data loss when an incorrect password is used at encryption. We therefore propose a new protocol that directly targets the problem of boostrapping encryption from a single password: distributed password-authenticated symmetric key encryption (DPaSE).

DPaSE offers strong security and usability, such as protecting the user's password against online and offline attacks, and ensuring message privacy and ciphertext integrity as long as at least one server is honest. We formally define the desired security properties in the UC framework and propose a provably secure instantiation. The core of our protocol is a new type of OPRF that allows to extend a previous partially-blind-query with a follow-up request and will be used to blindly carry over passwords across evaluations and avoid online attacks. Our (proof-of-concept) implementation of DPaSE uses 10 exponentiations at the user, 4 exponentiations and 2 pairings at each server, takes 105.58 ms to run with 2 servers and has a server throughput of 40 encryptions per second.

1 Introduction

Outsourcing storage to cloud providers is not only a common approach in enterprise settings, but is also widely appreciated by end users relying on services such as Dropbox, Google Drive, iCloud or Microsoft OneDrive to manage their personal data. With data breaches happening on a daily basis, it is essential that personal data kept in such cloud storage must be protected accordingly. The prevalent approach is to trust the cloud with properly encrypting the data, where the service provider controls access to the respective encryption keys via standard user authentication, mostly relying on username-password authentication. Clearly, such a solution crucially relies on the honesty of the service provider who can otherwise gain plaintext access to the users' data.

A different approach is let the user already encrypt the data before storing it in the cloud, which is offered e.g., by Tresorit [tre] or Mega [meg]. Therein a user client is locally encrypting the data and only uploads ciphertexts to the cloud. The cryptographic keys are either generated and stored directly by the user client, or (re)-derived from a human-memorizable password that the user enters into the local client. The former provides strong security guarantees, but is cumbersome to use as it relies on users' being able to manage and securely store cryptographic keys. The latter provides (roughly) the same convenience and usability as standard cloud provides as it does not require secure storage on the user side, but is inherently vulnerable to so-called offline attacks: Since encryption keys are derived from a low-entropy password, a corrupt service provider or an attacker gaining access to the ciphertexts, can attempt to decrypt the files by guessing the user's password.

*This work was partly supported by the EU Horizon 2020 research and innovation programme under grant agreement No 786725 OLYMPUS, by the German Research Foundation DFG - SFB 1119 - 236615297 (CROSSING Project S7), by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the *National Research Center for Applied Cybersecurity ATHENE*.

Encryption via Password-Authenticated Secret Sharing While enabling secure yet convenient key management for end users is still a largely unsolved problem in the real world, (partial) cryptographic solutions do already exist. A promising direction is that of password-authenticated secret sharing (PASS/PPSS) which allows a user to recover a strong secret that is shared among n servers when she can enter the correct password [BJSL11, CLLN14, JKK14, JKKX16, JKKX17]. In contrast to end users, servers can easily maintain strong cryptographic keys which is leveraged by PASS to thwart offline attacks against the password (and consequently on the shared secret key) if at least one, or a certain threshold, of the servers is not compromised. Relying on the assistance of external servers also allows to limit the impact of online attacks or password / device compromise, e.g., if a user suspects that her password might have been compromised, she can alert the servers to lock her account and disallow any further queries to recover the shared secret key.

Thus, a seemingly straightforward solution for password-based encryption is to rely on PASS to derive the required encryption keys from a user's password. However, this naïve approach comes again with a trade-off between security and usability: Either the same secret-shared key is used to encrypt all the user data, or file-specific keys are generated. The former is clearly better in terms of user convenience but provides only very limited security, as even a short breach of the user's device allows an adversary to recover the secret key that allows to decrypt *all* her files. Having dedicated keys for every file avoids this critical vulnerability but has several other disadvantages:

Wrong passwords can lead to permanent data loss: When the user derives a dedicated PASS key for every file, there is no simple and generic way to ensure that encryption is actually done with the *correct* password: Sharing a secret in PASS does not come with any password verification, thus any password is valid in that step. Only when the secret is *recovered* there is an implicit check that the entered password was correct. Thus, if the user uses a "wrong" password (i.e., not the one she wants to use to protect all her files online) when encrypting a new file, she ends up implementing her own cryptolocker, as decryption will fail when she later uses the "correct" password again. While the risk of simple typos can be reduced by asking the user to enter her password twice before encryption, more subtle mistakes such as entering a password from a different context, or not noticing that the keyboard layout has changed are impossible to detect. Unless the user makes the same mistake at decryption again, the plaintext files cannot be recovered and might be lost permanently.

Servers cannot recognize online guessing attempts: Another problem with individual keys is that they provide a much bigger surface for online guessing attacks. Most PASS schemes do not allow the *servers* to check whether the password used during recovery was correct. An exception is the Memento-protocol [CLLN14], which however is several magnitudes slower than the recent line of OPRF-based PASS protocols [JKK14, JKKX16, JKKX17]. Thus, when the user has to run a dedicated PASS protocol for every file it wants to enc- or decrypt, then accessing a larger chunk of data can easily trigger several thousand PASS requests. This can be used to camouflage online guessing attacks against an honest user's password, where an adversary is using a different password guess in each PASS request. Since the servers in efficient PASS protocols cannot recognize whether the provided passwords are correct, they cannot distinguish between legitimate queries and such online attacks.

Leakage of access pattern: If a dedicated PASS-key is used for every object, the servers will inherently learn which files the user wants to access with every decrypt query. Such leakage is known to have devastating effects on the user's privacy [IKK12, LMP18].

1.1 Our Contributions

In this work we address the problem of usable yet strongly secure password-based encryption. Acknowledging the fact that password-based schemes are inherently insecure in single-party/server settings, we follow the multi-server approach of PASS and propose a new type of protocol: distributed password-authenticated symmetric encryption (DPaSE).

DPaSE allows users to securely and conveniently encrypt and decrypt their data while relying only on a single password and the assistance of n servers. We formally define the desired security and privacy guarantees of DPaSE in the UC-framework and provide an efficient realization based on a new type of a partially-oblivious PRF that supports correlated evaluations of blind inputs; which we believe to be of independent interest.

In brief, DPaSE must provide the following functionality and security.

Correct Encryption: To avoid the aforementioned cryptolocker-by-accident situations, we start by modeling the account setting that is common in cloud-based storage solutions. That is, the user first creates an account with the n servers that is protected with a password pw. If she later wants to encrypt a file, she again enters a password pw' and encryption will only succeed when her password was correct. Likewise, also decryption requires using the correct password.

Security against Offline Attacks: As long as at least one server is honest, the encrypted data (or rather the underlying password) cannot be offline attacked. And even if eventually *all* servers are corrupted, they cannot decrypt the data immediately but must still perform an offline attack on each password – thus when users have chosen strong passwords, their data remains secure.

Security against Online Attacks: To detect and prevent online guessing attacks, the servers learn which user is trying to encrypt or decrypt, and whether her entered password was correct. When an honest server has recognized suspicious behaviour or was alerted by the user herself, it will enforce user-specific rate limiting or even fully block a certain account.

Obliviousness: While we rely on the servers for achieving strong security guarantees, we do not want them to infer anything about the user's behaviour and access patterns on her data. That is, servers should not learn anything about the files (plain- or ciphertext) the user wants to access, i.e., they cannot even tell whether they get decryption requests for the same or two different ciphertexts.

Authenticated Encryption: We also want to provide security against tampering attacks, where an adversary plants wrong information into the outsourced storage. Thus, unless the adversary knows the user's password (and is assisted by all server) it must be infeasible to create valid ciphertext.

Security Model in UC Framework. We formally define these properties in the Universal Composability (UC) framework, which is known to allow for the most realistic model for how users (mis)handle passwords. In game-based security models, users choose their passwords at random from known distributions and are assumed to behave perfectly, i.e., never make a typo when using a password. This clearly does not reflect reality, where users share or re-use passwords, and make mistakes when typing them. The UC framework models that much more naturally as therein the environment provides the passwords. Thus, a UC security notion guarantees the desired security properties without making any assumptions regarding the passwords' distributions or usages.

Efficient DPaSE **Protocol.** After formalizing DPaSE as an ideal functionality $\mathcal{F}_{\mathsf{DPaSE}}$ we present an efficient protocol that provably realizes $\mathcal{F}_{\mathsf{DPaSE}}$. The high-level idea of the protocol is very simple: To create an account, the user derives a signing key $(upk, usk) \leftarrow \mathsf{OPRF}(K, uid, pw)$ from her username and password, where the OPRF key K is split among the n servers and the evaluation reveals the username to the servers to later allow for user-specific rate limiting. The servers store (uid, upk) upon registration.

To encrypt a file, the user again enters uid, pw' and starts by re-running the steps from account creation to recover her signing key pair (upk, usk). She then signs a fresh nonce with usk and sends it to the servers who verify it against the stored upk, thereby verifying that $pw = pw'$. If the password is correct, the user and server engage in a follow-up OPRF evaluation where an object-specific encryption key is derived. The OPRF evaluation thereby "reuses" the previously entered uid, pw' to ensure that the actual encryption keys are also bound to the user' identity and correct password. This prevents users from accidentally encrypting data under a wrong password. To ensure obliviousness, the object (oid) for which the key is derived is hidden in the evaluation.

Decryption works almost analogously to encryption, verifying the password and – if correct – recovering the object-specific encryption key via the distributed OPRF. The generated ciphertexts and decryption proceeds also include checks to guarantee the desired ciphertext integrity.

Extendable Distributed Partially-Oblivious PRF. The core of our DPaSE protocol is a new type of OPRF that we believe to be of independent interest for many password-based applications. So far, OPRFs have been designed as *single*-evaluation primitives[1] that can either be fully or partially-blind. Thus, the user sends a (partially) blind query, and receives a single output related to that input. What we need for DPaSE though is an OPRF that "remembers" the blindly provided password from a previous query and re-uses it in a follow-up evaluation: we need to perform a dedicated password check and also want to ensure that encryption is done with the same password that was verified. We model that as an extension query, where a second OPRF query re-uses the blinded input from a previous request. This extension feature is required on top of *partial*-blindness (as the *uid*'s must be a known input to all parties), *verifiability* and the *distributed* setting. We formalize the desired properties of such an extendable OPRF in the UC framework and propose a secure instantiation.

Our OPRF construction is based on the classical double-hash DH scheme, basically combining all tricks that have been used in this context into a single scheme. The challenge thereby is that our second OPRF call which blindly carries over the input from the first call now has *three* inputs: the non-blind part $(x_{\mathsf{pub}} = uid)$, and two blinded values, namely the blinded $(x_{\mathsf{priv},1} = pw)$ from the previous evaluation and the new input $(x_{\mathsf{priv},2} = oid)$. Previous partially-blind OPRFs deal with two inputs only x_{pub} and x_{priv} which are mostly combined through a pairing, with the final PRF being of the form $\mathsf{H}_T(e(\mathsf{H}_1(x_{\mathsf{priv}}), x_{\mathsf{pub}})^K, x_{\mathsf{priv}})$ [ECS+15, BFH+20]. In our construction, we will already need both "slots" of the pairing to combine the two blinded inputs, and therefore must find a different place to include the public input. We take inspiration from [JKR19] and replace the direct use of the server's secret key K by $K' \leftarrow \mathsf{F}(K, uid)$ where F is a standard PRF. Thus, overall our new OPRF computes the output for an extended query as $\mathsf{H}_T(e(\mathsf{H}_1(x_{\mathsf{priv},1}), \mathsf{H}_2(x_{\mathsf{priv},2})^{\mathsf{F}(K, x_{\mathsf{pub}})}, x_{\mathsf{priv},1}, x_{\mathsf{priv},2}))$.

[1] With the exception of [Leh19] which proposes OPRF with batch evaluations. There one blinded input can be evaluated under several keys, which is orthogonal to our problem: we have a single key but want to extend one blinded input with a follow-up query.

The first (non-extended) query, just consisting of x_{pub} and $x_{\mathsf{priv},1}$ has the same form and simply sets $x_{\mathsf{priv},2} = 1$.

This construction allows us to combine three values into a single evaluation, but this extendability feature comes for a price. First, relying on exponents that are derived from a standard PRF $K' \leftarrow \mathsf{F}(K, uid)$ only allows for a distributed, but not threshold protocol. The distributed version simply considers the additive combination of all K' as the implicit overall secret key (per x_{pub}). Second, there are currently no efficient proofs that allow to check whether the final OPRF output has been computed correctly – which again stems from the use of the standard PRF to derive the OPRF secret key share. As we will need verifiability in our DPaSE protocol, we must assume that the servers in the OPRF are at most honest-but-curious.

Implementation and Evaluation. Instantiating DPaSE with our OPRF, yields an efficient scheme that requires 10 exponentiations at the user, 4 exponentiations and 2 pairings at each server. We further provide a proof-of-concept implementation of Π_{DPaSE} which respectively takes 45.96 ms for an account creation and 105.58 ms for each encryption and decryption with 2 servers; currently the implementation has a server throughput of 40 encryption or decryption requests per second.

1.2 Other Related Work

Apart from Password-Authenticated Secret Sharing we already discussed, there are further approaches that share some similarities but differ considerably in the overall setting:

Password-hardened encryption (PHE) [LER+18] targets a related setting, where a user outsources the handling of encryption keys to an external server and a rate limiter. The rate limiter can be implemented in a threshold version [BEL+20] to further enhance PHE's security. However, in PHE the frontend server is fully trusted, as it learns the user's password and keys. We cannot simply subsume the server in PHE as being part of the user in our setting, as the server also keeps secure state: Our goal is to bootstrap secure encryption from a password alone, without requiring the user to manage secret keys herself.

The recently proposed protocol for Updatable Oblivious Key Management [JKR19] also relies on a OPRF to derive file-specific encryption keys with the help of a (single) external server for increased security. Their work focuses on an enterprise setting for storage systems though, i.e., it relies on strong authentication between the client (that wants to enc/decrypt) and the server that holds the OPRF key. The challenge in our work was how to achieve such oblivious key management without strong authentication and relying only on a weak password.

Likewise, the DiSE protocol [AMMR18] and its more robust version [WH20] for distributed symmetric encryption consider strong authentication only. In these protocols, a group of n parties jointly controls encryption keys under which ciphertexts for the group get encrypted. The secret key material is split among the group and *any* member of the group can request decryption of ciphertexts which is again done jointly by all member. DiSE implicitly – yet crucially – relies on strong authentication to ensure that only valid members of the group can make such requests, whereas we want only a single user to enc/decrypt her files from a password. The authenticity checks in the enc/decryption process of our protocol are strongly inspired by DiSE though.

Finally, the PESTO protocol [BFH+20] for distributed single sign-on (SSO) relies on a similar idea of first deriving a strong key pair from a distributed OPRF in order to let a user authenticate to a number of servers. The overall application is different though, SSO vs. encryption, and consequently also the desired functionality and security are different. In particular, PESTO guarantees no security when all servers are corrupt, whereas our scheme still falls back to standard password protection in that case, i.e., the servers must perform offline attacks on all users passwords.

2 Preliminaries

2.1 Bilinear Groups

Definition 2.1 (Asymmetric Pairing). *Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be cyclic groups of order p with generators g_1, g_2, g_T, respectively. Furthermore, let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an efficiently computable non-degenerate function such that $\forall a, b \in \mathbb{Z}_p :\ e(g_1^a, g_2^b) = g_T^{ab}$. Then e is called an* asymmetric pairing. *$\mathbb{G} = (p, g_1, g_2, g_T, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ is called an* asymmetric bilinear group setting, *or* bilinear group *for short.*

We define an assumption on $\mathbb{G}$ by the following experiment with algorithm $\mathcal{A}$:

```
***Experiment*** $\mathsf{Exp}^{\mathbb{G}}_{\mathcal{A},\mathsf{Gapom\text{-}BDH}}(\lambda)$:
  $k \xleftarrow{\$} \mathbb{Z}_p$, $q_C \leftarrow 0$, $X_1 \leftarrow \emptyset$, $X_2 \leftarrow \emptyset$.
  $\{(x_i, y_i, z_i)\}_{i \in [\ell]} \leftarrow \mathcal{A}^{O_{\mathbb{G}\text{-}1}, O_{\mathbb{G}\text{-}2}, O_{\mathsf{D\text{-}help}}, O_{\mathsf{C\text{-}help}}}(\mathbb{G}, g_2^k)$
  **return** 0 if
    $0 \leq \ell - 1 < q_C$ or
    $\exists i \in [\ell] : (x_i \notin X_1 \vee y_i \notin X_2)$ or
    $\exists i, j \in [\ell], i < j : (x_i = x_j \wedge y_i = y_j)$
  **return** 1 if $\forall i \in [\ell] : e(x_i, y_i)^k = z_i$ and 0 otherwise.
```

where the experiment uses the following oracles

```
$\underline{O_{\mathbb{G}\text{-}\mathsf{r}}()}$
  return $\perp$ if $r \notin \{1, 2\}$
  $x \xleftarrow{\$} \mathbb{G}_r$
  $X_r \leftarrow X_r \cup \{x\}$
  return $x$
```

```
$\underline{O_{\mathsf{C\text{-}help}}(m)}$
  return $\perp$ if $m \notin \mathbb{G}_T$.
  $q_C \leftarrow q_C + 1$
  return $m^k$
```

```
$\underline{\mathcal{O}_{\mathsf{D\text{-}help}}(m, w, m', w')}$
    return $\perp$ if either $m, w, m', w'$ not in $\mathbb{G}_T$
    return 1 if $\log_m(w) = \log_{m'}(w')$ and else 0
```

To win, $\mathcal{A}$ needs to find pairs $(x, y, e(x,y)^k)$ without querying $e(x,y)$ to $\mathcal{O}_{\mathsf{C\text{-}help}}$ and where $\mathcal{A}$ could not rerandomize previous such pairs as it does not know the discrete logarithm of any x, y (enforced by sampling them at random using $\mathcal{O}_{\mathbb{G}\text{-r}}$). $\mathcal{A}$ is equipped with a DDH oracle $\mathcal{O}_{\mathsf{D\text{-}help}}$ in the group $\mathbb{G}_T$. The game $\mathsf{Gapom\text{-}BDH}$ follows the definition in [ECS+15].

Definition 2.2 (Gap One-More BDH Assumption)**.** *Let* $\lambda \in \mathbb{N}$ *be a security parameter and* $\mathbb{G} = (p, g_1, g_2, g_T, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ *be a bilinear group with* $\log(p) = poly(\lambda)$*, then we then say that the Gap One-More Bilinear Diffie-Hellman (Gapom-BDH) assumption holds for* $\mathbb{G}$ *if for all PPT adversaries* $\mathcal{A}$ *there is a negligible function* $\mathsf{negl}\cdot$ *such that* $\Pr[\mathsf{Exp}^{\mathbb{G}}_{\mathcal{A},\mathsf{Gapom\text{-}BDH}}(\lambda) = 1] \leq \mathsf{negl}\lambda$.

2.2 Digital Signatures

Definition 2.3 (Signature Schemes)**.** *A* signature scheme SIG *is a triple of algorithms* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ *with the following properties. On input the security parameter* λ*, the randomized* key generation algorithm Gen *outputs a key pair* (pk, sk)*. On a message* $m \in \{0,1\}^*$ *and a secret key sk, the randomized* signing algorithm Sign *outputs a signature* σ*. On input a public key pk, a message* $m \in \{0,1\}^*$*, and a signature* σ*, the deterministic* verification algorithm Verify *outputs* 1 *if the signature is correct or* 0 *otherwise.*

We require that the scheme satisfies *correctness*, i.e., for all $m \in \{0,1\}^*$ and (pk, sk) output by Gen it holds that: $\mathsf{Verify}(pk, m, \mathsf{Sign}(sk, m)) = 1$. Additionally, the scheme needs to satisfy *unforgeability under chosen message attacks (UF-CMA-security)*, i.e., after learning signatures for q number of adaptively chosen messages $\{m_1, ..., m_q\} \in \mathcal{M}$, it should be impossible to find a signature/message pair (σ, m) s.t. $\mathsf{Verify}(pk, m, \sigma) = 1$ and $m \notin \{m_1, ..., m_q\}$.

3 Verifiable Extendable Distributed Partially-Oblivious PRF

Our DPaSE construction relies on a new type of oblivious PRF (OPRF) that allows for extension queries and which we believe to be of interest for password-based protocols in general. In this section, we define this new type of OPRF and present a provably secure construction.

An OPRF is an interactive protocol between at least one user and one server. The server holds the key K of a pseudorandom function PRF, the user contributes the input x to the function. After the protocol run, the user holds the PRF evaluation at x, $\mathsf{PRF}_K(x)$. The obliviousness property demands that, while the server actively participated in the protocol, he did not learn anything about the value x he helped in evaluating the function for. On the other side, the user requires participation of the server to evaluate $\mathsf{PRF}_K()$ at any input. In a distributed OPRF, the key K is split among n servers.

Recently, there has been a flurry of OPRF constructions in the literature all featuring different (combinations of) properties on top of the above mentioned [JL09, JKK14, CL17, JKKX16, ECS+15, JKKX17, CL17, JKX18, BFH+20]. For constructing DPaSE, we require a new set of properties that we detail now. Our OPRF is called a *verifiable extendable distributed partially-oblivious PRF* (vedpOPRF).

Partial Obliviousness: The *obliviousness* property guarantees that the servers do not learn on which input ($x_{\mathsf{priv},1}$ and $x_{\mathsf{priv},2}$) the user wants to evaluate the function on. *Partial* obliviousness allows for an additional public part (x_{pub}) of the input.

Distribution: Obtaining a PRF value requires the active participation of all n servers. No subset of $n-1$ servers can evaluate the function themselves.

Verifiability: Outputs are guaranteed to be correct.

Extendability: After the user has provided an input $(x_{\mathsf{pub}}, x_{\mathsf{priv},1})$ and learned the corresponding output $\mathsf{PRF}_K(x_{\mathsf{pub}}, x_{\mathsf{priv},1})$, he can extend the query with a second blind input $x_{\mathsf{priv},2}$ upon which he receives $\mathsf{PRF}_K(x_{\mathsf{pub}}, x_{\mathsf{priv},1}, x_{\mathsf{priv},2})$ (in both cases the output is conditioned on the participation of all servers of course).

While the first three properties exist (individually) already, the concept of extendability of an OPRF is new. What is so special about this property that could not be achieved by simply evaluating the OPRF twice? The crucial difference is that an extendable OPRF guarantees that certain *blinded inputs are reused* in the second evaluation. With separate evaluation requests this cannot be guaranteed since blindings information-theoretically hide inputs and thus users can easily cheat. For DPaSE, we require such an OPRF to allow for dedicated password verification and ensuring that actual enc/decryption happens with the *same* password. We envision extendable OPRFs to be generally useful in protocols requiring more than one OPRF evaluation and where secret inputs of these single evaluations need to be correlated.

3.1 Ideal functionality for vedpOPRF

We define a verifiable extendable distributed partially-oblivious PRF in the Universal Composability framework [Can01] in terms of an ideal functionality $\mathcal{F}_{\mathsf{vedpOPRF}}$ in Figure 1. For brevity, we assume the following writing conventions.

- The functionality considers a specific session $sid = (S_1, \ldots, S_n, sid')$ and only accepts inputs from servers S_i that are contained in the sid.
- When the functionality is supposed to retrieve an internal record, but no such record can be found, then the query is ignored.

The functionality is parametrized by a security parameter λ. It interacts with servers $\mathcal{S} := \{S_1, \ldots, S_n\}$ (specified in the *sid*), arbitrary other parties and an adversary $\mathcal{A}$. $\mathcal{F}_{\mathsf{vedpOPRF}}$ maintains a table $T(x_{\mathsf{pub}}, x_{\mathsf{priv},1}, x_{\mathsf{priv},2})$ initially undefined everywhere, counters $\mathsf{ctr}[x_{\mathsf{pub}}]$ initially set to 0. $\mathcal{F}_{\mathsf{vedpOPRF}}$ sends all inputs to the adversary except for $x_{\mathsf{priv},1}, x_{\mathsf{priv},2}$.

Key Generation

- **On receiving** (KeyGen, sid) **from** S_i**:**
  - Ignore if the *sid* is marked `ready`.
  - If (KeyGen, sid) was received from all S_i, mark *sid* as `ready`, and give output $(\texttt{KeyConf}, sid)$ to all S_i.

Evaluation

- **On receiving** $(\texttt{EvalInit}, sid, qid, x_{\mathsf{pub}}, x_{\mathsf{priv},1})$ **from any party** U **(including** $\mathcal{A}$**):**
  - Record $(\texttt{eval}, sid, qid, U, x_{\mathsf{pub}}, x_{\mathsf{priv},1}, \perp)$, and send output $(\texttt{EvalInit}, sid, qid, x_{\mathsf{pub}})$ to every S_i.
- **On receiving** $(\texttt{EvalProceed}, R, sid, qid)$ **from** S_i **where** $R \in \{1,2\}$**:**
  - Retrieve record $(\texttt{eval}, sid, qid, U, x_{\mathsf{pub}}, x_{\mathsf{priv},1}, x_{\mathsf{priv},2})$, where $x_{\mathsf{priv},2} = \perp$ if $R = 1$, and $x_{\mathsf{priv},2} \neq \perp$ if $R = 2$.
  - If $(\texttt{EvalProceed}, R, sid, qid)$ has been received from all S_i, set $\mathsf{ctr}[x_{\mathsf{pub}}] \leftarrow \mathsf{ctr}[x_{\mathsf{pub}}] + 1$.
- **On receiving** $(\texttt{EvalFollow}, sid, qid, x_{\mathsf{priv},2})$ **from any party** U **(including** $\mathcal{A}$**):**
  - Retrieve record $(\texttt{eval}, sid, qid, U, x_{\mathsf{pub}}, x_{\mathsf{priv},1}, \perp)$ for (sid, qid, U).
  - Update record to $(\texttt{eval}, sid, qid, U, x_{\mathsf{pub}}, x_{\mathsf{priv},1}, x_{\mathsf{priv},2})$, and send output $(\texttt{EvalFollow}, sid, qid)$ to every S_i.
- **On receiving** $(\texttt{EvalComplete}, sid, qid)$ **from** $\mathcal{A}$**:**
  - Retrieve record $(\texttt{eval}, sid, qid, U, x_{\mathsf{pub}}, x_{\mathsf{priv},1}, x_{\mathsf{priv},2})$, only proceed if $\mathsf{ctr}[x_{\mathsf{pub}}] > 0$, set $\mathsf{ctr}[x_{\mathsf{pub}}] \leftarrow \mathsf{ctr}[x_{\mathsf{pub}}] - 1$.
  - If $T(x_{\mathsf{pub}}, x_{\mathsf{priv},1}, x_{\mathsf{priv},2})$ is undefined, then pick $\rho \overset{\$}{\leftarrow} \{0,1\}^{\lambda}$ and set $T(x_{\mathsf{pub}}, x_{\mathsf{priv},1}, x_{\mathsf{priv},2}) \leftarrow \rho$.
  - Output $(\texttt{EvalComplete}, sid, qid, x_{\mathsf{priv},2}, T(x_{\mathsf{pub}}, x_{\mathsf{priv},1}, x_{\mathsf{priv},2}))$ to U.

Figure 1: Ideal functionality $\mathcal{F}_{\mathsf{vedpOPRF}}$

- We assume private delayed outputs, meaning that the adversary can schedule their delivery but not read their contents beyond session and sub-session identifiers.

The functionality $\mathcal{F}_{\mathsf{vedpOPRF}}$ is inspired by functionalities from the literature and introduces extendability as a new OPRF feature. $\mathcal{F}_{\mathsf{vedpOPRF}}$ talks to arbitrary users and a fixed set of servers $S_1, \ldots, S_n$. Initially, all servers are required to call the KeyGen interface, to activate the functionality. Modeling an ideal PRF, $\mathcal{F}_{\mathsf{vedpOPRF}}$ chooses outputs at random, maintaining a function table $T()$ to ensure consistency. Implementing a partially-oblivious function, $\mathcal{F}_{\mathsf{vedpOPRF}}$ tells the servers public input x_{pub} before they have to decide about their participation in the request. Participation is signaled by calling (or not calling) `EvalProceed`. The adversary may also evaluate the function, but crucially requires participation of all servers as well. If all servers are corrupted, the adversary can freely evaluate the function by sending `EvalProceed` on behalf of all the corrupted servers. In order to allow for efficient protocols, we employ an "evaluation ticket" counter ctr[] allowing mixing-and-matching evaluations w.r.t the public input, as common for OPRF functionalities (see, e.g., [BFH$^+$20]). Also following the literature, we allow overwriting of inputs in adversarial evaluation requests in order to avoid the need to use extractable primitives only.

Our $\mathcal{F}_{\mathsf{vedpOPRF}}$ provides a new feature: it can be extended to output a second PRF value which is related to the first evaluation. This works as follows. A user obtains an evaluation on inputs $x_{\mathsf{pub}}, x_{\mathsf{priv},1}$ by calling `EvalInit` with session identifier *qid*. (The output is only generated if all servers participate and the adversary allows the output by calling `EvalComplete`, which is standard procedure for distributed OPRFs and we thus not elaborate here.) Afterwards, the user can provide a third input $x_{\mathsf{priv},2}$ via interface `EvalFollow`, using the still active session *qid*. $\mathcal{F}_{\mathsf{vedpOPRF}}$ outputs the function value at inputs $x_{\mathsf{pub}}, x_{\mathsf{priv},1}, x_{\mathsf{priv},2}$, ensuring that inputs $x_{\mathsf{pub}}, x_{\mathsf{priv},1}$ from the first evaluation are reused by looking them up using identifier *qid*.

3.2 Our vedpOPRF Construction

We now present our construction of a verifiable extendable distributed partially-oblivious PRF. Π_{vedpOPRF} computes the function

$$\mathsf{PRF}(K(x_{\mathsf{pub}}), x_{\mathsf{priv},1}, 1) = \mathsf{H}_T(x_{\mathsf{priv},1}, e(\mathsf{H}_1(x_{\mathsf{priv},1}), \mathsf{H}_2(1))^{K(x_{\mathsf{pub}})})$$

$$\mathsf{PRF}(K(x_{\mathsf{pub}}), x_{\mathsf{priv},1}, x_{\mathsf{priv},2}) =$$

$$\mathsf{H}_T(x_{\mathsf{priv},1}, x_{\mathsf{priv},2}, e(\mathsf{H}_1(x_{\mathsf{priv},1}), \mathsf{H}_2(x_{\mathsf{priv},2}))^{K(x_{\mathsf{pub}})})$$

with $K(x_{\mathsf{pub}}) \leftarrow \sum_{i=1}^{n} \mathsf{F}(k_i, x_{\mathsf{pub}})$ for (standard) PRF $\mathsf{F} : \{0,1\}^* \rightarrow \mathbb{Z}_q$, and k_i from F's key space is held by server S_i. x_{pub} denotes the public input and $x_{\mathsf{priv},1}, x_{\mathsf{priv},2}$ the private inputs. The function $e()$ denotes a pairing.

Setup and Key Generation: We require an asymmetric bilinear group $(g_1, g_2, g_T, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ and hash functions $\mathsf{H}_1 \colon \{0,1\}^* \rightarrow \mathbb{G}_1$, $\mathsf{H}_2 \colon \{0,1\}^* \rightarrow \mathbb{G}_2$, $\mathsf{H}_T \colon \{0,1\}^* \rightarrow \mathbb{G}_T$. We assume servers to choose keys $k_i \leftarrow \mathbb{Z}_q, i \in [n]$ at the beginning of the protocol.

Evaluation: Our PRF is essentially the "2Hash Diffie-Hellman" function [JKK14, JKKX16] $\mathsf{PRF}(k, x_{\mathsf{priv},1}) =$

$\mathsf{H}(x_{\mathsf{priv},1}, \mathsf{H}'(x_{\mathsf{priv},1})^k)$. Let us briefly explain how evaluating this function would work. A user *blinds* his input $x_{\mathsf{priv},1}$ with randomness r as $\mathsf{H}'(x_{\mathsf{priv},1})^r$ and sending this value to the server. The server sends back $\mathsf{H}'(x_{\mathsf{priv},1})^{rk}$, from which the user can compute $\mathsf{H}'(x_{\mathsf{priv},1})^k$ by exponentiation with $1/r$. This is enough for the user to compute $\mathsf{H}(x_{\mathsf{priv},1}, \mathsf{H}'(x_{\mathsf{priv},1})^k)$.

Partial obliviousness is now achieved as in Everspaugh et al. [ECS+15] by combining blinded private inputs as $e(\mathsf{H}_1(x_{\mathsf{priv},1})^r, \mathsf{H}_2(x_{\mathsf{priv},2})^k)$ using the pairing $e()$. Due to the bilinear property of $e()$ this is equal to $e(\mathsf{H}_1(x_{\mathsf{priv},1}), \mathsf{H}_2(x_{\mathsf{priv},2}))^{rk}$, which again allows the user to remove the blinding factor r. One can additively share k among all servers and let the client combine evaluation shares using the group operation in $\mathbb{G}_T$. The function is computed as $\mathsf{PRF}(k, x_{\mathsf{priv},1}, x_{\mathsf{priv},2}) = \mathsf{H}_T(x_{\mathsf{priv},1}, x_{\mathsf{priv},2}, e(\mathsf{H}_1(x_{\mathsf{priv},1}), \mathsf{H}_2(x_{\mathsf{priv},2}))^k)$.

For our new extendability property we require a PRF evaluated on three inputs. Fortunately, we can efficiently and securely augment the function given above with another input by "squeezing" it into the function's key. This technique is inspired by the work of Jarecki et al. [JKR19]. We set $K(x_{\mathsf{pub}}) := k \leftarrow \sum_{i \in [n]} \mathsf{F}(k_i, x_{\mathsf{pub}})$ for a (standard) PRF F and k_i being the servers' secret keys. One subtlety here occurs in the first evaluation on x_{pub} and $x_{\mathsf{priv},1}$ only. We cannot save the pairing evaluation and simply use $H_1(x_{\mathsf{priv},1})^k$ as k-dependent value instead: with this value, a user could compute arbitrary function evaluations on input $x_{\mathsf{priv},1}$ by himself by applying the pairing. We therefore let servers use a "dummy" value $H_2(1)$ and pair it with the user's blinded input $x_{\mathsf{priv},1}$.

A full formal description of our OPRF construction Π_{vedpOPRF} can be found in Figure 2. Its security is stated in the following theorem, for which a proof sketch can be found in Appendix A.

Theorem 3.1. *Let $\mathbb{G} = (p, g_1, g_2, g_T, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ be a bilinear group. If the Gapom-BDH assumption holds for $\mathbb{G}$ then the protocol Π_{vedpOPRF} given in Figure 2, with $\mathsf{H}_1, \mathsf{H}_2$ and H_T modeled as random oracles and* F *being a (standard) PRF, UC-emulates $\mathcal{F}_{\mathsf{vedpOPRF}}$ in the random oracle model assuming secure and server-side authenticated channels and honest-but-curious corruption of servers.*

On malicious security. As detailed above, our $\mathcal{F}_{\mathsf{vedpOPRF}}$ ensures that the PRF is always evaluated w.r.t the same key. This rules out protocols where servers can freely decide what key material to use in an evaluation. Let us note that it is quite common in the literature [JKKX17, JKX18, BFH+20] to relax this property by letting the OPRF functionality maintain different lists representing different PRF keys. The adversary can then determine which list is going to be used (in case of server corruption). And indeed, it turns out that our $\mathcal{F}_{\mathsf{vedpOPRF}}$ enforcing such consistency in keys is challenging to realize in the presence of malicious servers. The reason is that we cannot use standard techniques such as NIZK proofs of honest behavior with respect to some public server key (e.g., [ECS+15]), since our server keys are user-specific. Reliable distribution of such keys would involve frequent interaction with a trusted authority. Another inefficient way to obtain malicious security is to use a 3-linear map instead of a 2-linear map (pairing), together with a NIZK. The map would allow us to have (NIZK-compatible) *uid*-independent key shares simply by putting *uid* as third input parameter to the map. We choose not to give a maliciously secure protocol with such inefficient techniques, and rather leave the construction of an *efficient* maliciously secure verifiable extendable distributed partially-oblivious PRF as an open problem.

4 Distributed Password Authenticated Symmetric Encryption (DPaSE)

In this section we introduce distributed password-authenticated symmetric encryption (DPaSE). DPaSE is an interactive protocol between many users and a fixed set of n servers, where the servers assist users in conveniently and securely encrypting their data under a single password. DPaSE operates account-based: First, users register with a username and a single password at all servers. After account creation, the servers (blindly) assist users in encryption and decryption provided that they are using the correct password.

We recall the key security properties of DPaSE as already explained in more detail in Section 1.

Correct Encryption: DPaSE leverages the account-based concept to ensure that encryption can only succeed with the correct password.

Security against Offline Attacks: No message that is stored or send must allow offline attacks against the underlying password. This must hold as long as not all servers are corrupted. Even in such worst case scenario, messages remain protected by passwords, requiring servers to run a dictionary attack until they figure out the user's password.

Security against Online Attacks: With every enc/ decryption request, the servers learn the username and whether her password was correct. Servers can then refuse participation to block online attacks.

Obliviousness: Neither in registration nor during password verification the servers learn the user's password. Further, they cannot tell which objects are encrypted or decrypted, in order to hide access patterns on users' data.

Authenticated Encryption: Without knowing the correct password, it must be infeasible for an attacker to produce a valid ciphertext.

Our concrete scheme will leverage the servers mainly to (re)construct object-specific encryption keys, whereas the enc- and decryption happens locally at the user side. This might pose the question why we are modelling DPaSE as an encryption and not key management protocol. We have opted

USER U		SERVER S_i, holding k_i
On input $(\texttt{EvalInit}, sid, qid, x_{\mathsf{pub}}, x_{\mathsf{priv},1})$		
$r_1 \xleftarrow{\$} \mathbb{Z}_p$, $x_1 \leftarrow \mathsf{H}_1(x_{\mathsf{priv},1})^{r_1}$	$\xrightarrow{x_{\mathsf{pub}}, x_1}$	Output $(\texttt{EvalInit}, sid, qid, x_{\mathsf{pub}})$
		On input $(\texttt{EvalProceed}, 1, sid, qid)$
Upon receiving y_j from all $S_j \in \mathcal{S}$:	$\xleftarrow{y_i}$	$osk_i \leftarrow \mathsf{F}(k_i, uid)$, $y_i \leftarrow e(x_1, H_2(1))^{osk_i}$
$y \leftarrow \prod_{j\in[n]} y_j^{r_1^{-1}}$, $Y_1 \leftarrow \mathsf{H}_T(x_{\mathsf{priv},1}, y)$		
Output $(\texttt{EvalComplete}, sid, qid, \perp, Y_1)$		
On input $(\texttt{EvalFollow}, sid, qid, x_{\mathsf{priv},2})$		
$r_2 \xleftarrow{\$} \mathbb{Z}_p$, $x_2 \leftarrow \mathsf{H}_2(x_{\mathsf{priv},2})^{r_2}$	$\xrightarrow{x_2}$	Output $(\texttt{EvalFollow}, sid, qid)$
		On input $(\texttt{EvalProceed}, 2, sid, qid)$
Upon receiving y'_j from all $S_j \in \mathcal{S}$:	$\xleftarrow{y'_i}$	$y'_i \leftarrow e(x_1, x_2)^{osk_i}$
$y' \leftarrow \prod_{j\in[n]} {y'_j}^{r_1 r_2^{-1}}$, $Y_2 \leftarrow \mathsf{H}_T(x_{\mathsf{priv},1}, x_{\mathsf{priv},2}, y')$		
Output $(\texttt{EvalComplete}, sid, qid, x_{\mathsf{priv},2}, Y_2)$		

Figure 2: Protocol Π_{vedpOPRF}. We assume all messages to include sid, qid.

for capturing the full enc/decryption process to avoid similar misconceptions as with PASS, which was believed to be a suitable out-of-the-box tool for password-based encryption. Only with modelling and considering the full process this can be ensured.

4.1 An ideal functionality for DPaSE

We define DPaSE in terms of an ideal functionality $\mathcal{F}_{\mathsf{DPaSE}}$, which takes inputs of parties and hands them their securely computed outputs. $\mathcal{F}_{\mathsf{DPaSE}}$ abstracts away any protocol details and states only the required functionality and leakage and influence (i.e., attacks) allowed by an adversary.

We assume the same writing conventions as for $\mathcal{F}_{\mathsf{vedpOPRF}}$. In addition, we assume the adversary gets to acknowledge all inputs, but not learn their private content. For example, if the functionality receives input "$(\texttt{Encrypt}, sid, qid, x)$ from a party P" and "keeps x private", we assume that the functionality sends $(\texttt{Encrypt}, sid, qid, P)$ to the adversary and only processes the original input after receiving an acknowledgement from the adversary.

Our ideal functionality $\mathcal{F}_{\mathsf{DPaSE}}$ is depicted in Figure 3, with labeled instructions to enable easy matching to the explanations in this section. On a high level, $\mathcal{F}_{\mathsf{DPaSE}}$ is a password-protected lookup table for message-ciphertext pairs. Users can create new such pairs by first logging in to their account stored by $\mathcal{F}_{\mathsf{DPaSE}}$ with a username and password, and then encrypt a message of their choice, obtaining back the ciphertext. Decryption works in a similar fashion. $\mathcal{F}_{\mathsf{DPaSE}}$ stores a password for every registered user, and refuses service if a user does not remember his password correctly when he wants to encrypt or decrypt. In order to perform any of registration, encryption or decryption, $\mathcal{F}_{\mathsf{DPaSE}}$ requires participation of n distributed servers. We now describe the interfaces of $\mathcal{F}_{\mathsf{DPaSE}}$ in more detail. The functionality talks to arbitrary users and a fixed set of servers $S_1, \ldots, S_n$.

Account Creation: Any user can register with $\mathcal{F}_{\mathsf{DPaSE}}$ by calling its `Register` interface with a username *uid* and a password *pw*. If no account *uid* exists yet (R.3), $\mathcal{F}_{\mathsf{DPaSE}}$ informs all servers about the new registration request and the *uid*, but keeps the password private (R.4). Servers can now decide to participate in the registration by sending `ProceedRegister` to $\mathcal{F}_{\mathsf{DPaSE}}$. Only if all servers do so (PR.2), $\mathcal{F}_{\mathsf{DPaSE}}$ stores the account (uid, pw) and confirms the registration to the user. `ProceedRegister` inputs of servers are matched with their corresponding registration requests via subsession identifiers *qid*. Since those identifiers are unique, collecting servers' participation among different requests is prevented by $\mathcal{F}_{\mathsf{DPaSE}}$.

Encryption: A message encryption is initiated by a user sending an `Encrypt` request to $\mathcal{F}_{\mathsf{DPaSE}}$ which includes uid, pw and a message m. If account (uid, pw') exists (E.3), $\mathcal{F}_{\mathsf{DPaSE}}$ first informs all servers about an incoming request for *uid*, keeping the password as well as the message private (E.4). Only if all server agree to participate in this request for *uid* by using the `Proceed` interface for the corresponding subsession (P.2), $\mathcal{F}_{\mathsf{DPaSE}}$ continues the encryption request. By not giving away any information before, $\mathcal{F}_{\mathsf{DPaSE}}$ prevents offline attacks. $\mathcal{F}_{\mathsf{DPaSE}}$ now verifies that the provided password *pw* is equal to pw' stored with *uid* (P.2.3). All servers and the user are informed about the outcome of password verification by receiving either `PwdOK` or `PwdFail` (P.2.5). Being informed about failed password attempts and requests of *uid* in general allows servers to protect accounts against online guessing attacks: based on this information, they can decide to throttle requests by refusing to send `Proceed`, e.g., after 5 failed attempts within 1 minute. Such throttling is however decided by the application using $\mathcal{F}_{\mathsf{DPaSE}}$.

Finally, if verification was successful, the user obtains a ciphertext c from $\mathcal{F}_{\mathsf{DPaSE}}$. While c is adversarially chosen, we stress that, in an honest encryption procedure, the adversary only learns the length of the message from $\mathcal{F}_{\mathsf{DPaSE}}$ (P.2.2).

Thus, $\mathcal{F}_{\mathsf{DPaSE}}$ ensures that the ciphertext does not contain any information about m beyond its length. Further, $\mathcal{F}_{\mathsf{DPaSE}}$ ensures that no two encryption requests yield the same ciphertext by rejecting repeated ciphertexts sent by the adversary (P.2.2). $\mathcal{F}_{\mathsf{DPaSE}}$ stores the pair (m,c) together with *uid*, and sends the ciphertext to the user (P.2.7). Handing out the (fresh) ciphertext only if the correct password was provided ensures correct encryption. For this, note that for honestly registered accounts (which would not have $pw = \perp$) $\mathcal{F}_{\mathsf{DPaSE}}$ prevents the adversary from influencing the "verification bit" b in encryption procedures in any way.

Decryption: A user initiates a decryption procedure by calling the `Decrypt`interface of $\mathcal{F}_{\mathsf{DPaSE}}$ with uid, pw and a ciphertext c. It is instructive to note that $\mathcal{F}_{\mathsf{DPaSE}}$ does not give out any information about ciphertexts it is not explicitly queried for, and thus cannot be used as a storage for ciphertexts. $\mathcal{F}_{\mathsf{DPaSE}}$ informs servers about a request for *uid*, keeping password and ciphertext private (D.2). Similar to an encryption procedure, all servers are required to call `Proceed` in order for $\mathcal{F}_{\mathsf{DPaSE}}$ to continue with password verification (P.2). However, $\mathcal{F}_{\mathsf{DPaSE}}$ does not inform servers about which ciphertext should be decrypted, in order to hide access patterns on user data. We choose to not even inform servers about the type of request - encrypt or decrypt - in order to allow also for protocols where password verification requests do not yet reveal what a user wants to do. Coming back to the decryption procedure, $\mathcal{F}_{\mathsf{DPaSE}}$ now verifies the password (P.2.3). In case of success and if $\mathcal{F}_{\mathsf{DPaSE}}$ finds a record (uid, m, c), the message m is given to the requesting user (P.2.7). By storing *uid* along with message-ciphertext pairs and revealing m only if *uid*'s password was provided in the decryption query, $\mathcal{F}_{\mathsf{DPaSE}}$ enforces authenticated encryption.

Adversarial Interfaces: As explained before, we let $\mathcal{A}$ determine ciphertexts as common for functionalities modeling symmetric encryption. However, ciphertexts cannot depend on the message beyond its length, since $\mathcal{F}_{\mathsf{DPaSE}}$ ensures that the adversary is oblivious of messages to be encrypted (E.2 and P.2.2). $\mathcal{A}$ may also influence password verification in the following ways. First, modeling DoS attacks, we allow $\mathcal{A}$ to make individual servers believe that password verification failed even though the password might have been correct. This attack is carried out by setting $b_{S_i} \leftarrow 0$ for the corresponding server S_i (P.2.2 and P.2.4 , "otherwise" case). Second, $\mathcal{A}$ may make servers believe that password verification suceeded even when a wrong password was used, but only for accounts belonging to the adversary. $\mathcal{F}_{\mathsf{DPaSE}}$ marks a *uid* `corrupted` if this is the case, i.e., if a corrupted user performed a successful password verification with respect to username *uid* (P.2.1). The adversary then can fake successful password verification towards S_i by setting $b_{S_i} \leftarrow 1$ (P.2.4 , "Else, if" case). The motivation is that for such corrupted accounts we have to assume that the adversary knows all secrets. It is then plausible that he can compute whatever proof a protocol requires to convince servers of knowledge of the correct password.

We further weaken $\mathcal{F}_{\mathsf{DPaSE}}$ by allowing the adversary to start, e.g., an encryption request without yet knowing what message to decrypt, and under which password. Technically, this is enabled by $\mathcal{F}_{\mathsf{DPaSE}}$ accepting overwrite requests in adversarial records (R.1 , E.1 and D.1). While this does not constitute a meaningful attack for real-world applications (we stress that the adversary is only allowed to change inputs for his own requests, not for the ones of honest users), it allows for efficient realizations of $\mathcal{F}_{\mathsf{DPaSE}}$ such as ours based on oblivious pseudo-random functions and random oracles.

A special case: all servers corrupted. We want to highlight which guarantees $\mathcal{F}_{\mathsf{DPaSE}}$ gives in this worst case scenario. In any DPaSE protocol with n servers storing a somehow shared information about the user's password, these servers can always throw their data together, guess a password and run the password verification procedure of the DPaSE protocol to learn whether the guess was correct. This is unavoidable unless we involve more parties (such as an external password hardening service), which is not the scope of this work. However, we require that this is the best possible attack on the user's password when all servers are corrupted: they have to invest some computation to test each of their password guesses. This way, users with strong passwords will remain safe even in this worst case scenario. Since $\mathcal{F}_{\mathsf{DPaSE}}$ enforces authenticated encryption by revealing messages only if the correct password was supplied (P.2.3 and P.2.7) - regardless of how many servers are corrupted - not only passwords but also encrypted data of users with strong passwords remain secure.

4.2 A DPaSE protocol Π_{DPaSE}

We now present our DPaSE protocol Π_{DPaSE}. The detailed formal description can be found in Figure 4. Π_{DPaSE} uses hash functions $\mathsf{H}\colon \{0,1\}^* \to \{0,1\}^\lambda$ and $\mathsf{H\text{-}PRG}\colon \{0,1\}^\lambda \to \{0,1\}^*$, a signature scheme SIG and $\mathcal{F}_{\mathsf{vedpOPRF}}$ as ideal building block. The main principle of Π_{DPaSE} is that the servers assist the user in turning his (low-entropy, but unique) authentication data, i.e., username and password, into various (high-entropy) cryptographic keys. Those keys are subsequently used for proving knowledge of the password to servers, and to encrypt or decrypt the data. We describe the three phases of Π_{DPaSE}, account creation, encryption and decryption, in more detail in the below.

Account Creation: To create an account, a user derives a signing key pair (usk, upk) from its username *uid* and password *pw*. For this, $\mathcal{F}_{\mathsf{vedpOPRF}}$ is queried with inputs uid, pw by the user, yielding $Y \leftarrow \mathsf{PRF}(k, (pw, 1, uid))$ if all servers participate in the evaluation. Partial blindness ensures that servers learn *uid* but not *pw*. The user then computes $(usk, upk) \leftarrow \mathsf{SIG.Gen}(Y)$, sends *upk* to all servers and can afterwards delete the key pair. Servers are required to store (uid, upk).

The functionality is parametrized by a security parameter λ. It interacts with servers $\mathcal{S} := \{S_1,...,S_n\}$ (specified in the *sid*), as well as arbitrary users and an adversary $\mathcal{A}$.

Account Creation

- **On receiving** $(\texttt{Register}, sid, qid, uid, pw)$ **from** U **(including** $\mathcal{A}$**):**
  - R.1 If there already exists a record $(\texttt{register}, qid, U, uid, pw')$, if $U = \mathcal{A}$ then overwrite pw' with pw and if $U \neq \mathcal{A}$ then ignore the query.
  - R.2 Keep *pw* private
  - R.3 Proceed only if *no* record $(\texttt{account}, uid, *)$ exists. Create record $(\texttt{register}, qid, U, uid, pw)$.
  - R.4 Send a delayed output $(\texttt{Register}, sid, qid, uid)$ to all $S_i \in \mathcal{S}$.

- **On receiving** $(\texttt{ProceedRegister}, sid, qid, uid)$ **from server** S_i**:**
  - PR.1 Retrieve record $(\texttt{register}, qid, U, uid, pw)$.
  - PR.2 If $(\texttt{ProceedRegister}, sid, qid, uid)$ has been received from all S_i:
    - $*$ Record $(\texttt{account}, uid, pw)$; if U is corrupted mark *uid* `corrupted`. Send a delayed output $(\texttt{Registered}, sid, qid, uid)$ to U.

Encryption and Decryption

- **On receiving** $(\texttt{Encrypt}, sid, qid', uid, m, pw')$ **from party** U **or** $\mathcal{A}$**:**
  - E.1 If there already exists a record $(*, qid', U, uid, m', pw)$, if $U = \mathcal{A}$ then overwrite it with $(\texttt{enc}, qid', U, uid, m, pw')$ and if $U \neq \mathcal{A}$ then ignore the query.
    // $\mathcal{A}$ can change mode, password and message in his own requests
  - E.2 Keep `Encrypt`, m and pw' private, instead add `Request` tag and $\ell(m)$ in message to $\mathcal{A}$
  - E.3 Proceed only if a record $(\texttt{account}, uid, pw)$ exists. Create record $(\texttt{enc}, qid', U, uid, m, pw')$.
  - E.4 Send a delayed output $(\texttt{Request}, sid, qid', uid)$ to all $S_i \in \mathcal{S}$.

- **On receiving** $(\texttt{Decrypt}, sid, qid', uid, c, pw')$ **from party** U **or** $\mathcal{A}$**:**
  - D.1 If there already exists a record $(*, qid', U, uid, pw)$, if $U = \mathcal{A}$ then overwrite it with $(\texttt{dec}, qid', U, uid, c, pw')$ and if $U \neq \mathcal{A}$ then ignore the query.
  - D.2 Keep `Decrypt`, c and pw' private, instead add `Request` tag and $\ell(c)$ in message to $\mathcal{A}$
  - D.3 Proceed only if a record $(\texttt{account}, uid, pw)$ exists. Create record $(\texttt{dec}, qid', U, uid, c, pw')$.
  - D.4 Send a delayed output $(\texttt{Request}, sid, qid', uid)$ to all $S_i \in \mathcal{S}$.

- **On receiving** $(\texttt{Proceed}, sid, qid')$ **from server** S_i**:**
  - P.1 Retrieve records $(\texttt{mode}, qid', U, uid, \mathsf{obj}, pw')$ and $(\texttt{account}, uid, pw)$ with $\texttt{mode} \in \{\texttt{enc}, \texttt{dec}\}$.
  - P.2 If $(\texttt{Proceed}, sid, qid')$ has been received from all S_i:
    - P.2.1 If U corrupted and $pw == pw'$ then mark *uid* `corrupted`.
      // DoS attacks: $\mathcal{A}$ can prevent or sometimes even fake password confirmation (send $b_{S_i} = 0$ or $b_{S_i} = 1$)
    - P.2.2 Send $(\texttt{Complete}, sid, qid', pw == pw')$ to $\mathcal{A}$ and receive back $(\texttt{Complete}, sid, qid', b_{S_1}, \ldots, b_{S_n}, c)$. Abort if $\texttt{mode} = \texttt{enc}$ and c has been sent before.
    - P.2.3 If $pw = \perp$ then set $b \leftarrow \perp$; otherwise set $b \leftarrow (pw = pw')$.
    - P.2.4 For $i \in [n]$, if $pw = \perp$ set $b'_i \leftarrow 0$. Else, if U and *uid* corrupted then set $b'_i \leftarrow b_{S_i}$, otherwise set $b'_i \leftarrow b \wedge b_{S_i}$.
    - P.2.5 For $i \in [n]$, if $b'_i = 0$ output $(\texttt{PwdFail}, sid, qid')$ and otherwise output $(\texttt{PwdOK}, sid, qid')$ to all S_i.
    - P.2.6 If $b = 0$ output $(\texttt{PwdFail}, sid, qid')$ to U.
    - P.2.7 If $b = 1$ then
      - · If $\texttt{mode} = \texttt{dec}$ and a record (uid, m, obj) exists, output $(\texttt{Plaintext}, sid, qid', m)$ to U.
      - · If $\texttt{mode} = \texttt{enc}$ then store (uid, obj, c) and output $(\texttt{Ciphertext}, sid, qid', c)$ to U.

Figure 3: Ideal functionality $\mathcal{F}_{\mathsf{DPaSE}}$ for distributed password-authenticated symmetric encryption. For easy access to explanations we use highlighted numbering in both figure and text.

Encryption: Users are required to provide their correct password whenever they want to encrypt (or decrypt) any data. This is now straightforward: as in account creation, the user calls $\mathcal{F}_{\mathsf{vedpOPRF}}$ with inputs (uid, pw'), receiving PRF value Y_1 if again all servers participate in the PRF evaluation. The user now computes a signing key pair from Y_1, signs part of the transcript (we mention that the identifier of this encryption session, qid', is globally unique) and sends the resulting signature σ_U to each server. Servers will accept (i.e., output `PwdOK`) only if σ_U is a verifying signature under *upk* stored with *uid*, which happens if and only if $pw = pw'$. Of course, this verification technique only works if servers reliably learn *uid* used in the PRF compuation, which is ensured by the partial obliviousness of $\mathcal{F}_{\mathsf{vedpOPRF}}$.

Symmetric encryption in Π_{DPaSE} is simply a one-time pad, with an object-specific encryption key which is computed again from a PRF value and with the help of all servers. But now, Π_{DPaSE} crucially relies on the extendability of the PRF to ensure correct and authenticated encryption of message m under encryption randomness ρ. Namely, the key is computed from $\mathsf{H}(m,\rho)$ and uid, pw that successfully verified before. Note that this requires to evaluate the PRF on three inputs, while the two latter are reused from the password verification procedure detailed above. The extendability property of $\mathcal{F}_{\mathsf{vedpOPRF}}$ allows this by calling `EvalFollow` with input $\mathsf{H}(m,\rho)$, still using the identifier qid' of the ongoing encryption session. The user obtains $Y_2 \leftarrow \mathsf{PRF}(k,(pw,\mathsf{H}(m,\rho),uid))$ from $\mathcal{F}_{\mathsf{vedpOPRF}}$.

To encrypt, Y_2 is XORed with (m,ρ) (applying H-PRG first to account for differences in lengths). The resulting ciphertext is augmented with $\mathsf{H}(m,\rho)$. The reason for appending this auxiliary information will become apparent below.

Decryption: In order to compute a decryption key, a user first has to successfully pass password verification. This is done in the exact same way as for an encryption request (in fact, in our protocol, servers cannot distinguish an encryption request from a decryption request). Computation of the decryption key is also done the exact same way as in encryption – now it becomes apparent why $com \leftarrow \mathsf{H}(m,\rho)$ is required to be part of the ciphertext. The user decrypts m,ρ by XORing the decryption key with the first part of the ciphertext. Finally, the user verifies correct decryption by recomputing com from m,ρ. While PRF computation is verifiable when using $\mathcal{F}_{\mathsf{vedpOPRF}}$, the latter check is still required since otherwise faulty ciphertexts (where the com contains another message) would decrypt faithfully and users would recover data that they never encrypted.

4.3 Security of Π_{DPaSE}

For analyzing the security of Π_{DPaSE} we assume that honest users delete all protocol values such as Y_1, Y_2, usk after performing an encryption or decryption, i.e., upon closing a subsession. Further, we assume that within ongoing subsessions (the identifier qid indicates one such session) an honest user does not get corrupted. This seems reasonable given the fact that, in reality, the time between password verification and encryption (or decryption) will be only very few seconds.

Theorem 4.1. *The protocol Π_{DPaSE} given in Figure 4 with $\mathsf{H}, \mathsf{H\text{-}PRG}$ modeled as random oracles and $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ an EUF-CMA-secure signature scheme UC-emulates $\mathcal{F}_{\mathsf{DPaSE}}$ in the $\mathcal{F}_{\mathsf{vedpOPRF}}$-hybrid random oracle model w.r.t static malicious server corruption and assuming server-side authenticated and secure channels.*

Proof Sketch. A detailed description of simulated cases can be found in Table 3 in the Appendix. Here, we only give a brief overview of different simulation aspects and particular challenges.

Simulation of honest servers. Since servers do not obtain any secret input that is kept from the adversary, simulating honest servers is quite trivial: the simulator $\mathcal{S}$ just follows the server's protocol.

Simulate honest user without password. First note that the password influences the outputs of the (deterministic) PRF. To know whether former PRF values have to be reused as output (i.e., in case of a correct password), it is enough for the simulator to learn whether password verification was successful. Fortunately, $\mathcal{S}$ learns this information from $\mathcal{F}_{\mathsf{DPaSE}}$ on time (via `(Complete,...)` message) before having to commit to any $\mathcal{F}_{\mathsf{vedpOPRF}}$ output.

Extraction of corrupted user's secrets. Since any user, even a corrupted one, needs to use $\mathcal{F}_{\mathsf{vedpOPRF}}$ in order to obtain a key, $\mathcal{S}$ can extract a corrupted user's password and message or ciphertext from his inputs to $\mathcal{F}_{\mathsf{vedpOPRF}}$. Another way to see this is that, while usage of $\mathcal{F}_{\mathsf{vedpOPRF}}$ simplifies our DPaSE simulator's life in this case, the burden is on the protocol realizing $\mathcal{F}_{\mathsf{vedpOPRF}}$. This protocol has to ensure that secrets can be extracted from adversarial messages.

Three different ways to encrypt or decrypt. The simulation is complicated by the fact that $\mathcal{Z}$ can initiate, e.g., an encryption procedure either via an honest user and then recompute the symmetric decryption key via either a corrupted user or via $\mathcal{Z}$'s adversarial interface at $\mathcal{F}_{\mathsf{vedpOPRF}}$. Knowing only the ciphertext so far, $\mathcal{S}$ now needs to produce the symmetric key without knowing the plaintext it should decrypt to. However, all honest servers need to agree to help $\mathcal{Z}$ in computation of the symmetric key. $\mathcal{S}$ can use the server's agreement to obtain the plaintext message from $\mathcal{F}_{\mathsf{DPaSE}}$, compute the key linking this message with the ciphertext and send it to $\mathcal{Z}$ (for this, $\mathcal{S}$ has to program the random oracle H-PRG to point to the key). □

On security against malicious servers. In the previous theorem stating security of Π_{DPaSE}, we handle malicious server corruptions. However, since we use as a building block our Π_{vedpOPRF} which provides security only against honest-but-curious servers, our overall construction is only secure against such mild corruptions. We stress that, by proving the previous Theorem 4.1 w.r.t malicious corruptions, we do not want to create a false impression. Rather, by this we attempt to isolate the difficulty in achieving malicious security: if one manages to UC-realize $\mathcal{F}_{\mathsf{vedpOPRF}}$ w.r.t malicious server corruptions, with Theorem 4.1 one automatically obtains a DPaSE protocol secure against such corruptions.

For now, we obtain the following corollary by combining Theorems 4.1 and 3.1.

Corollary 4.2. *Let $\mathbb{G} = (p, g_1, g_2, g_T, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e)$ be a bilinear group and $\mathsf{H}_1, \mathsf{H}_2, \mathsf{H}_T, \mathsf{H}, \mathsf{H\text{-}PRG}$ hash functions as described in Π_{vedpOPRF} and Π_{DPaSE}, modeled as random oracles. If the Gapom-BDH assumption holds for $\mathbb{G}$, $\mathsf{SIG} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ is an EUF-CMA-secure signature scheme*

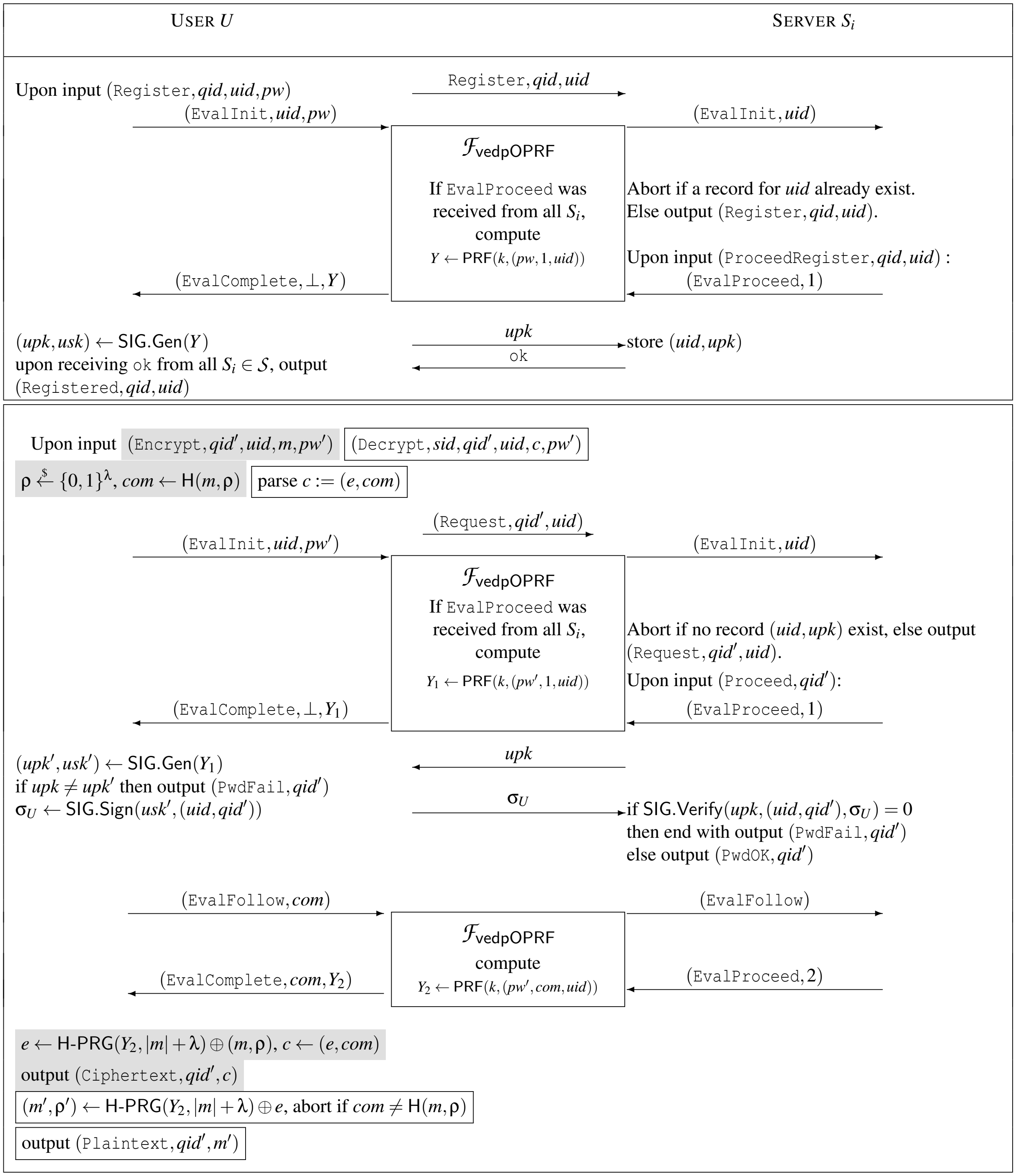
USER U
SERVER S_i
Upon input (Register, qid, uid, pw)
Register, qid, uid
(EvalInit, uid, pw)
(EvalInit, uid)
$\mathcal{F}_{\mathsf{vedpOPRF}}$
If EvalProceed was received from all S_i, compute
$Y \leftarrow \mathsf{PRF}(k, (pw, 1, uid))$
Abort if a record for uid already exist.
Else output (Register, qid, uid).
Upon input (ProceedRegister, qid, uid) :
(EvalProceed, 1)
(EvalComplete, $\perp, Y$)
$(upk, usk) \leftarrow \mathsf{SIG.Gen}(Y)$
upon receiving ok from all $S_i \in \mathcal{S}$, output (Registered, qid, uid)
upk
ok
store (uid, upk)
Upon input (Encrypt, qid', uid, m, pw') (Decrypt, sid, qid', uid, c, pw')
$\rho \xleftarrow{\$} \{0,1\}^\lambda$, $com \leftarrow \mathsf{H}(m, \rho)$
parse $c := (e, com)$
(Request, qid', uid)
(EvalInit, uid, pw')
(EvalInit, uid)
$\mathcal{F}_{\mathsf{vedpOPRF}}$
If EvalProceed was received from all S_i, compute
$Y_1 \leftarrow \mathsf{PRF}(k, (pw', 1, uid))$
Abort if no record (uid, upk) exist, else output (Request, qid', uid).
Upon input (Proceed, qid'):
(EvalProceed, 1)
(EvalComplete, $\perp, Y_1$)
$(upk', usk') \leftarrow \mathsf{SIG.Gen}(Y_1)$
if $upk \neq upk'$ then output (PwdFail, qid')
$\sigma_U \leftarrow \mathsf{SIG.Sign}(usk', (uid, qid'))$
upk
σ_U
if $\mathsf{SIG.Verify}(upk, (uid, qid'), \sigma_U) = 0$
then end with output (PwdFail, qid')
else output (PwdOK, qid')
(EvalFollow, com)
(EvalFollow)
$\mathcal{F}_{\mathsf{vedpOPRF}}$
compute
$Y_2 \leftarrow \mathsf{PRF}(k, (pw', com, uid))$
(EvalComplete, com, Y_2)
(EvalProceed, 2)
$e \leftarrow \mathsf{H\text{-}PRG}(Y_2, |m| + \lambda) \oplus (m, \rho)$, $c \leftarrow (e, com)$
output (Ciphertext, qid', c)
$(m', \rho') \leftarrow \mathsf{H\text{-}PRG}(Y_2, |m| + \lambda) \oplus e$, abort if $com \neq \mathsf{H}(m, \rho)$
output (Plaintext, qid', m')

Figure 4: Our protocol Π_{DPaSE} using a signature scheme SIG, Π_{vedpOPRF} protocol and hash functions $\mathsf{H}, \mathsf{H\text{-}PRG}$. Top box shows registration, bottom box shows encryption and decryption. Gray instructions are only executed in encryption, $\boxed{\text{framed}}$ ones only in decryption. Each encryption and decryption query has to use a fresh subsession identifier qid'.

and F *a (standard) PRF, then* Π_{DPaSE} *with* $\mathcal{F}_{\mathsf{vedpOPRF}}$ *instantiated by* Π_{vedpOPRF} *UC-emulates* $\mathcal{F}_{\mathsf{DPaSE}}$ *in the random oracle model w.r.t static honest-but-curious server corruption and assuming server-side authenticated and secure channels.*

5 Evaluation & Comparison

In this section we report on the efficiency of our scheme. We first compare our Π_{DPaSE} protocol with similar password-based schemes by counting the number of exponentiations

Scheme	Setting	#(Exponentiations + Pairings)	
		user (in [LER$^+$18] rate limiter)	(per) server
PASS-based Enc (Dec) [ECS$^+$15]	(t,n)	6 exps $(= 2\mathbb{G}_1 + 2\mathbb{G}_2 + 2\mathbb{G}_T)$ +1 pairing	3 exps $(= 2\mathbb{G}_1 + 1\mathbb{G}_T)$ +1 pairing
PHE [LER$^+$18]	rate limiter and server	7 exps (in $\mathbb{G}$)	10 exps (in $\mathbb{G}$)
DPaSE (Our Work)	(n,n)	10 exps $(= 2\mathbb{G}_1 + 2\mathbb{G}_2 + 4\mathbb{G}_T + 2\mathbb{G}_{\mathsf{p\text{-}256}})$	4 exps $(= 2\mathbb{G}_T + 2\mathbb{G}_{\mathsf{p\text{-}256}})$ +2 pairings

Table 1: Comparison of Π_{DPaSE} with other password-based encryption services, where the exponentiations are counted per group $\mathbb{G}$, $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$, the pairing is mapped as $\mathbb{G}_1 \times \mathbb{G}_2 : \to \mathbb{G}_T$ and $\mathbb{G}_{\mathsf{p\text{-}256}}$ represents the prime group in the ECDSA signature scheme secp256r1.

per group and pairings, being the most expensive operations of such protocols. The related schemes are the PASS scheme [ECS$^+$15], which gives a partially-oblivious thershold PRF and the password hardened encryption (PHE) from [LER$^+$18]. We stress that PASS has a number of usability and security drawbacks when used for encryption for multiple files, and PHE does consider a rather different setting between a (trusted) server and rate limiter. See Section 1 for a more detailed discussion. We still believe that comparing these works helps to evaluate the efficiency of our scheme.

Benchmarks We carried out a proof-of-concept implementation of our Π_{DPaSE} protocol and report preliminary benchmarks on the same. We implement in Java, and use the MIRACL - AMCL library for the pairing computation and exponentiation operations. We use the Boneh-Lynn-Shacham pairing with 461 bit curves for the pairing $\mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ in Π_{vedpOPRF}, ECDSA with sec256r1 as the user's signature scheme SIG, SHA-512 as the underlying hash function and the Java's inbuilt KeyPairGenerator class for user key pair generation $\mathsf{SIG.Gen}$. The elements in groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ are implemented using single exponentiation operations with the respective group generators. For implementing the standard PRF function F in Π_{vedpOPRF}, we used Java's inbuilt HMAC-SHA-256.

We measured our implementation on a machine running a Intel Core i7-7500U series CPU with 4 virtual CPUs, 16 GiB of RAM. We simulated the behavior of the user and the servers connected through java's serializable sockets. We do not report on the latency of the communication between the user and the server and rather focus on the local computations. A Π_{DPaSE} protocol run between a user and 2 servers took 45.96 milliseconds (ms) and 105.58 ms for *account creation* and an *encryption* request respectively, while using 5 servers the same took 99.36 ms and 243.86 ms respectively. This leads to processing 40 encryption or decryption requests per second for a server. The timing can be further optimized by supporting multi-threading computations.

6 Conclusion and Open Questions

In this paper, we introduce and construct a distributed password-authenticated symmetric encryption protocol (DPaSE), which lets users log in with only a password to perform symmetric encryption and decryption. DPaSE is useful in "mobile" scenarios where users do not have secure storage to securely maintain encryption keys. DPaSE is designed to offer strong provable security guarantees, such as protecting the user's password against on-line and off-line attacks, and message privacy and ciphertext integrity even if all servers are corrupted. Regarding usability, DPaSE protects users from accidentally encrypting their data with faulty passwords.

Our construction uses an oblivious pseudo-random function (OPRF) twice: first, to let the user turn her password into high-entropy authencation data, and second, to let the user compute a symmetric key. To ensure the strong guarantees mentioned above, we require an OPRF with special properties (extendable, verifiable, distributed and only partially oblivious). We give a construction for such an OPRF, which we believe is of independent interest as a new building block for password-based cryptographic protocols. We provide proof-of-concept implementations for our DPaSE construction (including our OPRF construction) and compare efficiency to related protocols in the literature. Our protocol provides only little overhead, scales well in the number of users and servers, and features provable security under standard bilinear discrete-log based assumptions in the random oracle model.

An interesting future direction is the construction of a *threshold* version of DPaSE, where only an arbitrary subset of all servers is required to participate in each user request. This would improve usability of the protocol, since users would not have to wait for answers of busy servers. However, our user-specific OPRF keys seem to rule out usage of standard techniques for threshold protocols.

Finally, security in the presence of malicious servers would be enabled by constructing a maliciously secure verifiable extendable distributed partially-oblivious PRF. Alternatively, for ensuring correct encryption it seems to be sufficient to have servers use the same keys in both OPRF evaluations. This flavor of verifiability in our OPRF seems to be achievable with standard techniques. Although key switching between different requests of a specific user would not significantly weaken but clutter the description of $\mathcal{F}_{\mathsf{DPaSE}}$, we decide to present the more secure and cleaner version here, and leave the slightly weaker but maliciously secure version as future work.

References

[AMMR18] Shashank Agrawal, Payman Mohassel, Pratyay Mukherjee, and Peter Rindal. DiSE: Distributed symmetric-key encryption. In *ACM CCS 18*, pages 1993–2010. ACM Press, 2018.

[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, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 409–424. ACM, 2020.

[BFH$^+$20] Carsten Baum, Tore K. Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. Pesto: Proactively secure distributed single sign-on, or how to trust a hacked server. IEEE European Symposium on Security and Privacy, 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 11*, pages 433–444. ACM Press, October 2011.

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

[CL17] Jan Camenisch and Anja Lehmann. Privacy-preserving user-auditable pseudonym systems. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 269–284. IEEE, 2017.

[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.

[CLN15] Jan Camenisch, Anja Lehmann, and Gregory Neven. Optimal distributed password verification. In Indrajit Ray, Ninghui Li, and Christopher Kruegel:, editors, *ACM CCS 15*, pages 182–194. ACM Press, October 2015.

[ECS$^+$15] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. pages 547–562, 2015.

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*. The Internet Society, February 2012.

[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 *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 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.

[JKR19] Stanislaw Jarecki, Hugo Krawczyk, and Jason K. Resch. Updatable oblivious key management for storage systems. In *ACM CCS 19*, pages 379–393. ACM Press, 2019.

[JKX18] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. LNCS, pages 456–486. Springer, Heidelberg, 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.

[Leh19] Anja Lehmann. Scrambledb: Oblivious (chameleon) pseudonymization-as-a-service. *Proc. Priv. Enhancing Technol.*, 2019(3):289–309, 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, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 1405–1421. USENIX Association, 2018.

[LMP18] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy*, pages 297–314. IEEE Computer Society Press, 2018.

[meg] Mega: Secure cloud storage and communication.privacy by design. https://mega.nz/.

[tre] Tresorit: Cloud storage + end-to-end encryption. https://tresorit.com/security/encryption.

[WH20] Xunhua Wang and Ben Huson. Robust distributed symmetric-key encryption. *IACR ePrint*, 2020:1001, 2020.

A Proof sketch of Theorem 3.1

In this section, we analyze the security of our protocol Π_{vedpOPRF} with respect to our new $\mathcal{F}_{\mathsf{vedpOPRF}}$. For this, we assume that users delete all protocol values such as r_1, r_2, y and y' after outputting both PRF evaluations. Further, we assume that within ongoing subsessions (the identifier *qid* indicates one subsession) an honest user does not get corrupted.

We further assume static honest-but-curious server corruptions. Essentially, a corrupted server's inputs and outputs are handled by the adversary, while its code is still executed as in the protocol instruction.

Proof sketch. Our proof combines aspects of the OPRF proofs given in [ECS$^+$15], [CLN15] and [BFH$^+$20], which all follow initial ideas of [JKKX17].

In Table 2 we provide details of the simulation of our distributed verifiable partially-oblivious PRF when interacting with $\mathcal{F}_{\mathsf{vedpOPRF}}$. The rows of the table consider all possible corruption scenarios, while the columns focus on different tasks of the simulator.

Usage of random oracles The random oracles are used in different ways: either $\mathcal{S}$ choses outputs such that he knows their discrete logarithms, or he observes queries to the oracle, or he programs the oracle's output to match other values from the simulation. More detailed, $\mathcal{S}$ uses discrete logarithms of H_1 and H_2 outputs to generate U's randomness r upon learning U's private input x_{priv}. With all three oracles, observability is exploited (to learn $x_{\mathsf{priv},i}$ and x_{pub} values). And finally, H_T is programmed to PRF values generated by $\mathcal{F}_{\mathsf{vedpOPRF}}$. Cf. Table 2 for more details on the simulation of the oracles.

Simulating without secrets First, we note that simulation of servers is trivial since inputs towards servers do not carry any secrets. The simulation works by, for all $i \in [n]$, simply running the code of S_i on a k_i randomly chosen in the beginning.

Contrarily, simulation of honest users must work without knowing secret values $x_{\mathsf{priv},1}, x_{\mathsf{priv},2}$. Observe that, in our OPRF protocol, a user computes $\mathsf{PRF}(x_{\mathsf{pub}}, x_{\mathsf{priv},1}, x_{\mathsf{priv},2})$ by hashing $\mathsf{H}_T(x_{\mathsf{priv},1}, x_{\mathsf{priv},2}, y')$, where y' depends on all inputs and the server's keys. However, since a PRF is a deterministic function, $\mathcal{Z}$ might obtain $\mathsf{PRF}(x_{\mathsf{pub}}, x_{\mathsf{priv},1}, x_{\mathsf{priv},2})$ via two ways: computing the Hash on its own through a corrupted user, or by running evaluation via an honest user. To make things look consistent, the simulator needs to make sure the result is the same both ways. For this, he recognizes *consistent* queries to H_T made by $\mathcal{Z}$ using knowledge of all k_i. If a consistent query is detected, $\mathcal{S}$ programs the result to the same PRF value that $\mathcal{F}_{\mathsf{vedpOPRF}}$ would output. We show below that, if the Gapom-BDH assumption holds in the underlying group, $\mathcal{S}$ will always obtain a PRF value from $\mathcal{F}_{\mathsf{vedpOPRF}}$ for each consistent H_T query made by $\mathcal{Z}$.

The challenge of simulating extendable evaluation One technicality that stems from extendable evaluations is the order in which $\mathcal{Z}$ computes PRF values through a corrupted user. Essentially, $\mathcal{Z}$ can complete a full run of the protocol without querying H_T for the final PRF values, and then query them in an arbitrary order. Our simulation has to deal with all possible orders, where it is crucial that $\mathcal{S}$ can match queries belonging to each other (note that due to the determinstic nature of a PRF, we cannot simply include subsession identifiers *qid* in H_T inputs to allow for such matching).

Reduction to the one-more BDH assumption Our simulation relies on $\mathcal{F}_{\mathsf{vedpOPRF}}$ leaking specific PRF values to $\mathcal{S}$ and we specify in Table 2 an event `fail` in which $\mathcal{S}$ does *not* receive this leakage. We show that `fail` happens only with negligible probability if the Gapom-BDH assumption holds in $\mathbb{G}$ by constructing a successful attacker $\mathcal{B}$ exploiting `fail`. On a high level, the idea is as follows. If $\mathcal{F}_{\mathsf{vedpOPRF}}$ does not provide a PRF value for inputs $x_{\mathsf{pub}}, x_{\mathsf{priv},1}, x_{\mathsf{priv},2}$, then at least one honest server did not proceed the request. This corresponds to one missing evaluation share. Thus, $\mathcal{Z}$ submitting a hash query $\mathsf{H}_T(x_{\mathsf{priv},1}, y_1)$ or $\mathsf{H}_T(x_{\mathsf{priv},1}, x_{\mathsf{priv},2}, y_2)$ solves a hard problem by computing a consistent y_1 or y_2. Namely, $\mathcal{Z}$ provides either a CDH tuple $(x, y, e(x,y)^k)$ or a CDH tuple $(x, z, e(x,z)^k)$, where $x \leftarrow H_1(x_{\mathsf{priv},1}), y \leftarrow H_2(x_{\mathsf{priv},2})$ and $z \leftarrow H_2(1)$. $\mathcal{B}$ will use its oracles to detect consistency and to simulate the execution without knowing discrete logs and server keys.

As a preparatory step, we choose all values $\mathsf{F}(k_i, x_{\mathsf{pub}})$ for honest servers S_i truly at random, which goes unnoticed by the environment due to pseudorandomness of F. This is needed to ensure that embedding of a Gapom-BDH challenge does not change any distribution.

Let us now state the reduction in more detail. Since `fail` only occurs if at least one server is honest, w.l.o.g we assume that all but one server eventually get corrupted. $\mathcal{B}$ chooses $j \in [n]$ at random and aborts if S_j gets corrupted. We show now how $\mathcal{B}$ emulates the ideal execution using the oracles provided by the Gapom-BDH experiment.

As a warm up, let us first assume that $\mathcal{Z}$ always uses the same x_{pub} in all inputs. $\mathcal{B}$, on input $(\mathbb{G}, K)$, where $K = g_2^k$ for some secret k, stores a "DDH reference tuple" $(\texttt{ref}, g_1, g_2, e(g_1, K))$. $\mathcal{B}$ chooses random keys k_i for $i \neq j$ and implicitly sets $osk_j \leftarrow k - \sum_{i \neq j} \mathsf{F}(k_i, x_{\mathsf{pub}})$. Servers $S_i \neq S_j$ are simulated as in the protocol using keys k_i. Note that the distribution of osk_j does not change by this embedding since $\mathsf{F}(k_j, x_{\mathsf{pub}})$ was chosen uniformly at random in our preparatory step above.

$\mathcal{B}$ uses his oracles $\mathcal{O}_{\mathbb{G}\text{-}1}, \mathcal{O}_{\mathbb{G}\text{-}2}$ to answer $\mathcal{Z}$'s queries to H_1 and H_2, respectively. Whenever S_j receives input $(\texttt{EvalProceed}, R)$ with $R \in \{1,2\}$ and corresponding messages x_{pub}, x_1 or x_2 with $x_1 \in G_1, x_2 \in G_2$ sent to S_j from a corrupted user, $\mathcal{B}$ proceeds as follows: if $R = 1$, $\mathcal{B}$ obtains $z \leftarrow \mathcal{O}_{\mathsf{C\text{-}help}}(e(x_1, H_2(1))$, adds $(\texttt{sol}, x_1, H_2(1), z)$ to a list of CDH solutions replies to the user with $y_j \leftarrow z / e(x_1, H_2(1))^{\sum_{i \neq j} \mathsf{F}(k_i, x_{\mathsf{pub}})}$. The case $R = 2$ is handled the same way.

If $\mathcal{Z}$ queries $\mathsf{H}_T(x_{\mathsf{priv},1}, y)$, $\mathcal{B}$ retrieves record $(\texttt{ref}, g_1, g_2, Z)$ and submits $(e(g_1, g_2), z, e(H_1(x_{\mathsf{priv},1}), H_2(1)), y)$ to $\mathcal{O}_{\mathsf{D\text{-}help}}$. In case of receiving 1, y is correctly computed and $\mathcal{B}$ submits `EvalInit` to $\mathcal{F}_{\mathsf{vedpOPRF}}$ as described in the simulation. $\mathcal{B}$ adds $(\texttt{sol}, H_1(x_{\mathsf{priv},1}), H_2(1)), y)$ to the list of CDH solutions. Upon $\mathcal{F}_{\mathsf{vedpOPRF}}$ outputting a PRF value Y_T, $\mathcal{B}$ programs $\mathsf{H}_T(x_{\mathsf{priv},1}, y) := Y_T$. $\mathcal{B}$ proceeds analogously for queries $\mathsf{H}_T(x_{\mathsf{priv},1}, x_{\mathsf{priv},2}, y')$ made by $\mathcal{Z}$.

If even `fail` occurs, then $\mathcal{B}$ submits his list of CDH solutions $(\texttt{sol}, \ldots)$. In this case, this list contains one more non-trivial CDH solution that the number of queries to $\mathcal{O}_{\mathsf{C\text{-}help}}$ and thus $\mathcal{B}$ wins the Gapom-BDH experiment.

Now we lift the restriction to one x_{pub}. Let $osk_j^{x_{\mathsf{pub},\ell}}$ denote the secret key that is used by S_j in a session for $x_{\mathsf{pub},\ell}$. The issue with different such $x_{\mathsf{pub},\ell}$ values is that $\mathcal{Z}$ knows a relation between (the unknown) Gapom-BDH exponent k and $osk_j^{x_{\mathsf{pub},\ell}}$ namely that their difference is $\sum_{i \neq j} \mathsf{F}(k_i, x_{\mathsf{pub},\ell})$. To make $osk_j^{x_{\mathsf{pub},\ell}}$ pseudorandom from $\mathcal{Z}$' viewpoint again, $\mathcal{B}$ implicitly sets $osk_j^{x_{\mathsf{pub},\ell}} \leftarrow (k - \sum_{i \neq j} \mathsf{F}(k_i, x_{\mathsf{pub},\ell}))\alpha_\ell + \beta_\ell$ for $\alpha_\ell, \beta_\ell \leftarrow \mathbb{Z}_p$ uniformly at random. $\mathcal{B}$ stores extended DDH reference tuples of the form $(\texttt{ref}, g_1, g_2, e(g_1, K)^{\beta_\ell} \cdot e(g_1, g_2^{\alpha_\ell}), \alpha_\ell, \beta_\ell, x_{\mathsf{pub},\ell})$ (one for each $x_{\mathsf{pub},\ell}$). $\mathcal{B}$ proceeds as in the simulation with one x_{pub}, with two changes: first, $\mathcal{B}$ adds randomizing factors α_ℓ, β_ℓ to simulated shares y_j by setting $y_j \leftarrow z^{\alpha_\ell} / e(x_1, H_2(1))^{\sum_{i \neq j} \alpha_\ell \cdot \mathsf{F}(k_i, x_{\mathsf{pub}}) + \beta_\ell}$ s.t. $y_j = e(x_1, H_2(1))^{osk_j^{x_{\mathsf{pub},\ell}}}$. Second, $\mathcal{B}$ checks consistency w.r.t all `ref` tuples now using oracle $\mathcal{O}_{\mathsf{D\text{-}help}}$. Assume the oracle outputs 1 for tuple $(\texttt{ref}, g_1, g_2, *, \alpha_\ell, \beta_\ell, x_{\mathsf{pub},\ell})$ and query $\mathsf{H}_T(x_{\mathsf{priv},1}, y)$. Let $e \leftarrow e(\mathsf{H}_1(x_{\mathsf{priv},1}), \mathsf{H}_2(1))$. Then $\mathcal{B}$ adds $(\texttt{sol}, \mathsf{H}_1(x_{\mathsf{priv},1}), \mathsf{H}_2(1), (y / e^{\beta_\ell - \sum_{i \neq j} \mathsf{F}(k_i, x_{\mathsf{pub},\ell})})^{\alpha_\ell^{-1}})$ to the list of CDH solutions and proceeds with querying $\mathcal{F}_{\mathsf{vedpOPRF}}$ as before.

□

B Additional material on proof of Theorem 4.1

In Table 3 we provide the simulation of our DPaSE protocol Π_{DPaSE} for a simulator interacting with $\mathcal{F}_{\mathsf{DPaSE}}$. The rows of the table consider all possible corruption scenarios, while the columns focus on different tasks of the simulator.

	Transcript	Input/output of user U	Random oracles
All honest	Trivial due to usage of secure channels	To make sure that U gets output, $\mathcal{S}$ sends $(\texttt{EvalComplete}, sid, qid, \mathsf{hon})$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$ each time all S_i messages are delivered to U.	$\mathcal{S}$ perfectly simulates all random oracles, meaning that he chooses fresh and uniformly random values from the groups G_1, G_2 and G_T for the functions $\mathsf{H}_1, \mathsf{H}_2$ and H_T, respectively. For each function $\mathcal{S}$ maintains a list of the form (H_1, x, y) where $\mathsf{H}_1(x) = y$.
U and up to $n-1$ S_i corrupted	Choose $k_1, \ldots, k_n$ at random. Simulate server messages as in protocol (servers do not have secret inputs) using those key shares.	Input: upon U sending x_{pub}, x_1 $\mathcal{S}$ chooses $x_{\mathsf{priv},1} \leftarrow G_1$ at random and inputs $(\texttt{EvalInit}, sid, qid, x_{\mathsf{pub}}, x_{\mathsf{priv},1})$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$ so that servers get $\texttt{Eval}$ output for qid (note that the simulated private input cannot be recognized by $\mathcal{Z}$ since the output towards S_i does not contain any x_{priv} value). Similarly, $\mathcal{S}$ chooses a random $x_{\mathsf{priv},2} \leftarrow G_2$ upon U sending x_2 to provide input $\texttt{EvalFollow}$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$.	On $\mathsf{H}_T(x_{\mathsf{priv},1}, y_1)$ of $\mathcal{Z}$ (first such query), $\mathcal{S}$ only proceeds if $y_1 = e(H_1(x_{\mathsf{priv},1}), g_2)^{\Sigma \mathsf{F}(k_i, x'_{\mathsf{pub}})}$ for some x'_{pub} used before by some corrupted U. If there is a record $(*, x_{\mathsf{priv},1}, *, x'_{\mathsf{pub}}, Y''_1, *)$ ($\mathcal{Z}$ **already queried second Hash**) then $\mathcal{S}$ programs $(\mathsf{H}_T, (x_{\mathsf{priv},1}, y_1), Y''_1)$ and answers $\mathcal{Z}$'s H_T query with Y''_1. If there is no such record ($\mathcal{Z}$ **queries 1st Hash first**), then $\mathcal{S}$ queries $\mathcal{F}_{\mathsf{vedpOPRF}}$ with $(\texttt{EvalInit}, sid, qid', x'_{\mathsf{pub}}, x_{\mathsf{priv},1})$ with a fresh identifier qid'. $\mathcal{S}$ delays the output $(\texttt{EvalInit}, \ldots)$ to servers infinitely. Then, $\mathcal{S}$ sends $(\texttt{EvalComplete}, sid, qid', \mathsf{hon})$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$. This will let $\mathcal{F}_{\mathsf{vedpOPRF}}$ use the counter increased due to the simulated $x_{\mathsf{priv},1}$ and give $\mathcal{S}$ one PRF value $(\texttt{EvalComplete}, sid, qid', \perp, Y_1)$. $\mathcal{S}$ then programs $(\mathsf{H}_T, (x_{\mathsf{priv},1}, y_1), Y'_1)$ and answers $\mathcal{Z}$'s H_T query with Y'_1. $\mathcal{S}$ stores the tuple $(qid', x_{\mathsf{priv},1}, \perp, x'_{\mathsf{pub}}, Y'_1, \perp)$. If $\mathcal{S}$ does not receive any Y'_1 from $\mathcal{F}_{\mathsf{vedpOPRF}}$, we say that event $\texttt{fail}$ happens. On $\mathsf{H}_T(x_{\mathsf{priv},1}, x_{\mathsf{priv},2}, y_2)$ of $\mathcal{Z}$ (first such query), if $y_2 = e(H_1(x_{\mathsf{priv},1}), H_2(x_{\mathsf{priv},2}))^{\Sigma \mathsf{F}(k_i, x''_{\mathsf{pub}})}$ for some x''_{pub} used before by some corrupted U, then $\mathcal{S}$ looks for a recorded tuple $(qid'', x_{\mathsf{priv},1}, *, x''_{\mathsf{pub}}, *, *)$ for some qid''. If none is found ($\mathcal{Z}$ **queries 2nd Hash first**), $\mathcal{S}$ proceeds as above with inputting $(\texttt{EvalInit}, sid, qid'', x''_{\mathsf{pub}}, x_{\mathsf{priv},1})$ (for a fresh qid'') to obtain some Y''_1. Afterwards, and also in case such a record already exists ($\mathcal{Z}$ **already queried 1st Hash**) $\mathcal{S}$ sends $(\texttt{EvalFollow}, sid, qid'', x_{\mathsf{priv},2})$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$, infinitely delaying outputs to servers again. $\mathcal{S}$ sends $(\texttt{EvalComplete}, sid, qid'', \mathsf{hon})$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$. Upon receiving $(\texttt{EvalComplete}, sid, qid'', x_{\mathsf{priv},2}, Y''_2)$, then $\mathcal{S}$ programs $(\mathsf{H}_T, (x_{\mathsf{priv},1}, x_{\mathsf{priv},2}, y_2), Y''_2)$ and answers $\mathcal{Z}$'s H_T query with Y''_2. $\mathcal{S}$ stores the tuple $(qid', x_{\mathsf{priv},1}, x_{\mathsf{priv},2}, x''_{\mathsf{pub}}, Y''_1, Y''_2)$. If $\mathcal{S}$ does not receive any Y''_2 from $\mathcal{F}_{\mathsf{vedpOPRF}}$, we say that event $\texttt{fail}$ happens.
U honest, up to $n-1$ S_i corrupted	$\mathcal{S}$ simulates the user's messages towards the corrupted S_i by choosing $x_1 \leftarrow G_1$, $x_2 \leftarrow G_2$ at random. Note that x_{pub} is given to $\mathcal{S}$ by $\mathcal{F}_{\mathsf{vedpOPRF}}$.	Input: comes from $\mathcal{Z}$. Output: since even corrupted servers follow the protocol, the simulation works as in the first row.	$\mathcal{S}$ might learn a honest user's input $x_{\mathsf{priv},1}$ via a $\mathsf{H}_T(x_{\mathsf{priv},1}, y_1)$ or $\mathsf{H}_T(x_{\mathsf{priv},1}, x_{\mathsf{priv},2}, y_2)$ query by $\mathcal{Z}$ for an y_1 or y_2 that he can compute as follows: $\mathcal{S}$ looks for a record (x_{pub}, v, w) (see leftmost column in this row how such records are created) such that $y_1^{v^{-1}} = e(H_1(x_{\mathsf{priv},1}), g_2)^{\Sigma \mathsf{F}(k_i, x_{\mathsf{pub}})}$, or $y_1^{w^{-1}} = e(H_1(x_{\mathsf{priv},1}), H_2(x_{\mathsf{priv},2}))^{\Sigma \mathsf{F}(k_i, x_{\mathsf{pub}})}$. If such a record is found then $\mathcal{S}$ recovers or creates an H_1 entry $(\mathsf{H}_1, x_{\mathsf{priv},1}, g^z, z)$ and sets the client's randomness to $r_1 = v/z$ such that $x_1 = g^v = g^{zr_1}$ and the simulated x_1 (see first column) is consistent with his simulation of the RO H_1. r_2 is set analogously.
U honest, all S_i corrupted	Same as above	Same as above	The simulation is a simpler version of the "U and up to $n-1$ S_i corrupted" case above, since now $\mathcal{S}$ can let corrupted servers proceed requests. Thus, $\mathcal{S}$ can always use fresh qid identifiers to get PRF evaluations from $\mathcal{F}_{\mathsf{vedpOPRF}}$ to program them into H_T. We let $\mathcal{S}$ always send all necessary proceeds such that event $\texttt{fail}$ never happens.

Table 2: Simulation of Π_{vedpOPRF}.

	Transcript	Input/output of user U and server S_i	$\mathcal{F}_{\mathsf{vedpOPRF}}$ and random oracles $\mathsf{H}, \mathsf{H\text{-}PRG}$
All honest	*Summary: Simulate random messages due to secure channels. Discover when client and server abort using $pw == pw'$ info from $\mathcal{F}_{\mathsf{DPaSE}}$ and the simulated b_{S_i} bits.* Due to usage of secure channels, all messages look random. $\mathcal{S}$ however needs to know if to simulate a message or not. For registration, $\mathcal{S}$ always simulates all messages. For $\mathtt{Encrypt}$and $\mathtt{Decrypt}$, upon receiving $(\mathtt{Complete}, qid', b)$, $\mathcal{S}$ only sends the client's last message if $b = 1$ (since a real world client aborts if $b = 0$). In that case, $\mathcal{S}$ simulates S_i's last message only if S_i received $\mathtt{PwdOK}$ from $\mathcal{F}_{\mathsf{DPaSE}}$. $\mathcal{S}$ knows if that is the case due to knowing b and b_{S_i} which he computes himself (see simulation on the right).	*Summary: Acknowledge inputs/outputs according to $\mathcal{A}$'s scheduling of messages.* $\mathcal{S}$ acknowledges all inputs and outputs according to the message scheduling of $\mathcal{A}$. Choose a random ciphertext of length ℓ using uid and ℓ learned via message $\mathtt{Request}$ from $\mathcal{F}_{\mathsf{DPaSE}}$. If $\mathcal{A}$ delivers all n messages qid', uid unmodified then $\mathcal{S}$ sends $(\mathtt{Complete}, sid, qid, 1, \ldots, 1, c)$ to $\mathcal{F}_{\mathsf{DPaSE}}$. $\mathcal{S}$ delays outputs $\mathtt{PwdOK}/\mathtt{PwdFail}$ towards a server until $\mathcal{A}$ delivers the corresponding message.	If $\mathcal{Z}$ sends $(\mathtt{EvalInit}, qid, uid, pw)$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$ via $\mathcal{A}$ (we can assume this is the first $\mathtt{EvalInit}$ query with qid), if a corrupted U already sent $(\mathtt{Register}, qid, uid)$ to all S_i, then $\mathcal{S}$ inputs $(\mathtt{Register}, qid, uid, pw)$ to $\mathcal{F}_{\mathsf{DPaSE}}$. In case of $\mathcal{Z}$ switching to pw' by sending $(\mathtt{EvalInit}, qid', uid, pw')$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$ via $\mathcal{A}$, $\mathcal{S}$ sends $(\mathtt{Register}, qid, uid, pw')$ to $\mathcal{F}_{\mathsf{DPaSE}}$, but this time delays outputs $(\mathtt{Register}, qid, uid)$ towards all S_i infinitely. After sending $\mathcal{F}_{\mathsf{vedpOPRF}}$'s output y to $\mathcal{Z}$, if U sends some *upk not* obtained from $\mathsf{SIG.Gen}(y)$, $\mathcal{S}$ sends $(\mathtt{Register}, qid, uid, \perp)$ to $\mathcal{F}_{\mathsf{DPaSE}}$, infinitely delaying the $\mathtt{Register}$ outputs towards servers (to ensure honest servers always output $\mathtt{PwdFail}$ for this account). Otherwise, qid does not belong to any registration query, but $\mathcal{Z}$ wants to compute a key for encryption or decryption. $\mathcal{S}$ waits for $\mathcal{Z}$ to sent $(\mathtt{EvalFollow}, com)$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$ via $\mathcal{A}$. Let $c := (e, uid, com)$ denote the ciphertext containing com sent already by $\mathcal{S}$ to $\mathcal{F}_{\mathsf{DPaSE}}$, otherwise let $c \leftarrow \perp$. $\mathcal{S}$ sends $(\mathtt{Decrypt}, qid, uid, c, pw)$ to $\mathcal{F}_{\mathsf{DPaSE}}$. If $\mathcal{Z}$ now sends $\mathtt{EvalComplete}$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$ let y_2 denote the second $\mathcal{F}_{\mathsf{vedpOPRF}}$ output sent to $\mathcal{Z}$ by $\mathcal{S}$ via $\mathtt{EvalComplete}$. Otherwise, $\mathcal{Z}$ possibly switches to values pw', com' (by re-sending $\mathtt{EvalInit}$ and $\mathtt{EvalFollow}$ for same uid but fresh qid'' to $\mathcal{F}_{\mathsf{vedpOPRF}}$) before fetching the PRF evaluations. Then let $pw \leftarrow pw'$, $com \leftarrow com'$, y_2 the second PRF value sent to $\mathcal{Z}$ and $c := (e, uid, com)$ the already sent ciphertext containing com, or $c \leftarrow \perp$ if none was sent. Now that $\mathcal{Z}$ has committed to obtain a key for pw and com, we distinguish three cases. **Case 1: c was sent by $\mathcal{S}$ to $\mathcal{F}_{\mathsf{DPaSE}}$ in request of honest user.** $\mathcal{S}$ sends $(\mathtt{Decrypt}, qid, uid, c, pw)$ to $\mathcal{F}_{\mathsf{DPaSE}}$ and delays outputs $(\mathtt{Request}, qid, uid)$ towards all S_i infinitely. Finally, upon $\mathcal{F}_{\mathsf{DPaSE}}$ sending $(\mathtt{Complete}, qid, b)$, if $\mathcal{A}$ delivers all n messages qid, uid unmodified, $\mathcal{S}$ replies with $(\mathtt{Complete}, qid, b_{S_1}, \ldots, b_{S_n}, (e, uid, com))$, where $b_{S_i} \leftarrow 0$ if $\mathcal{Z}$ sends a non-verifying signature to S_i and $b_{S_i} \leftarrow 1$ otherwise. In case of receiving a plaintext m from $\mathcal{F}_{\mathsf{DPaSE}}$, $\mathcal{S}$ chooses ρ at random and programs $H(m, \rho) := com$. Before sending $\mathcal{F}_{\mathsf{vedpOPRF}}$ output y_2 to $\mathcal{Z}$ $\mathcal{S}$ programs $\mathsf{H\text{-}PRG}(y_2, \lvert m\rvert + \lambda) := e \oplus (m, \rho)$. **Case 2: $c = \perp$ ($\mathcal{S}$ never sent *com* to $\mathcal{F}_{\mathsf{DPaSE}}$).** $\mathcal{S}$ looks for record $((m, \rho), com)$ in H list, creating a random one if none exists. $\mathcal{S}$ sends $(\mathtt{Encrypt}, qid, uid, m, pw)$ to $\mathcal{F}_{\mathsf{DPaSE}}$ and delays outputs $(\mathtt{Request}, qid, uid)$ towards all S_i infinitely . Upon sending $\mathcal{F}_{\mathsf{vedpOPRF}}$'s output y_2 to $\mathcal{Z}$, $\mathcal{S}$ programs $\mathsf{H\text{-}PRG}(y_2, \lvert m\rvert + \lambda) := e \oplus (m, \rho)$ for a randomly chosen e. Finally, upon $\mathcal{F}_{\mathsf{DPaSE}}$ sending $(\mathtt{Complete}, qid, b)$, if $\mathcal{A}$ delivers all n messages qid, uid unmodified, $\mathcal{S}$ replies with $(\mathtt{Complete}, qid, b_{S_1}, \ldots, b_{S_n}, (e, uid, com))$, where $b_{S_i} \leftarrow 0$ if $\mathcal{Z}$ sends a non-verifying signature to S_i and $b_{S_i} \leftarrow 1$ otherwise. **Case 3: c was sent by $\mathcal{S}$ to $\mathcal{F}_{\mathsf{DPaSE}}$ as in case 2 above.** $\mathcal{S}$ does not provide any further input to $\mathcal{F}_{\mathsf{DPaSE}}$. Since all necessary records in $\mathsf{H}, \mathsf{H\text{-}PRG}$ and $\mathcal{F}_{\mathsf{vedpOPRF}}$ already exist, no additional programming is necessary.
Only U corrupted	*Summary: $\mathcal{S}$ follows Π_{DPaSE} with simulated k_i to simulate honest servers' messages.* $\mathcal{S}$ determines which messages to simulate the same way as above. But now $\mathcal{S}$ actually has to simulate cleartext messages of honest servers. Upon U sending message upk_i to S_i, $\mathcal{S}$ sends back $\mathtt{ok}$. On $(\mathtt{Request}, qid', uid)$ $\mathcal{S}$ sends back upk_i from S_i. Then, if $\sigma_{U,i}$ sent by corrupted U to S_i verifies w.r.t upk_i, $\mathcal{S}$ sets $b_{S_i} \leftarrow 1$.	In case U sent $(\mathtt{Register}, qid, uid)$ to all S_i and $(\mathtt{EvalInit}, qid, pw)$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$ $\mathcal{S}$ inputs $(\mathtt{Register}, qid, uid, pw)$ to $\mathcal{F}_{\mathsf{DPaSE}}$ on behalf of the corrupted U. After sending $\mathcal{F}_{\mathsf{vedpOPRF}}$'s output y to U, if U sends some *upk not* obtained from $\mathsf{SIG.Gen}(y)$, $\mathcal{S}$ sends $(\mathtt{Register}, qid, uid, \perp)$ to $\mathcal{F}_{\mathsf{DPaSE}}$ (to ensure honest servers always output $\mathtt{PwdFail}$), infinitely delaying the $\mathtt{Register}$ output towards all S_i. Otherwise, if U wants to compute a key and sends $(\mathtt{EvalFollow}, qid, com)$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$, he committed to obtain the key for pw and com. $\mathcal{S}$ now proceeds as in the case distinction in the rightmost column, but replacing every occurence of $\mathcal{Z}$ in the code with U.	
U and up to $n-1$ S_i corrupted	Same as above, but only w.r.t all honest S_i.	Same as above. Additionally, upon a corrupted S_i sending $(\mathtt{EvalProceed}, qid)$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$, $\mathcal{S}$ sends $(\mathtt{ProceedRegister}, qid)$ to $\mathcal{F}_{\mathsf{DPaSE}}$ on behalf of this S_i in case qid is a registration query, and $(\mathtt{Proceed}, qid)$ otherwise.	
U honest, up to $n-1$ S_i corrupted	*Summary: simulate OPRF usage of honest client without password.* When U registers with uid, $\mathcal{S}$ simulates $\mathcal{F}_{\mathsf{vedpOPRF}}$ without a password as input, choosing a fresh output y as response to the simulated U and storing $((\cdot, uid), y)$ in the list of PRF values. Upon U encrypting or decrypting with uid, $\mathcal{S}$ again simulates $\mathcal{F}_{\mathsf{vedpOPRF}}$ without a password, and postpones giving $\mathcal{F}_{\mathsf{vedpOPRF}}$ output to the simulated U unless it receives $(\mathtt{Complete}, qid, b)$ from $\mathcal{F}_{\mathsf{DPaSE}}$. If $b = 1$, $\mathcal{S}$ sets $y_1 \leftarrow y$, else chooses a random fresh y_1. $\mathcal{S}$ chooses a fresh y_2 and continues simulation of U with $\mathcal{F}_{\mathsf{vedpOPRF}}$ output y_1, y_2.	Same as in first row, but only w.r.t honest S_i. Additionally, upon a corrupted S_i sending $(\mathtt{EvalProceed}, qid)$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$, $\mathcal{S}$ sends $(\mathtt{ProceedRegister}, qid)$ to $\mathcal{F}_{\mathsf{DPaSE}}$ on behalf of this S_i in case qid is a registration query, and $(\mathtt{Proceed}, qid)$ otherwise.	Same as above. Additionally, if $\mathcal{Z}$ obtains a PRF evaluation on (pw, uid) for an honestly registered uid by sending $(\mathtt{EvalInit}, qid, uid, pw)$ to $\mathcal{F}_{\mathsf{vedpOPRF}}$ via $\mathcal{A}$, after all servers proceeded, $\mathcal{S}$ obtains $(\mathtt{Complete}, qid, b)$ and sets $\mathcal{F}_{\mathsf{vedpOPRF}}$'s output to $y_1 \leftarrow y$ in case of $b = 1$. Here, y denotes the PRF value sent to the simulated honest user who registered uid. $\mathcal{S}$ fills the corresponding PRF record with pw (see leftmost column). In case of $b = 0$ $\mathcal{S}$ chooses a random fresh y_1. $\mathcal{S}$ sends y_1 as $\mathcal{F}_{\mathsf{vedpOPRF}}$'s output to $\mathcal{Z}$.
U honest, all S_i corrupted		Same as above. Additionally, note that all servers may now jointly switch to a *upk* different from the simulated honest user's verification key. In this case, $\mathcal{S}$ lets U output $\mathtt{PwdFail}$.	

Table 3: Simulation of Π_{DPaSE}