Threshold Partially-Oblivious PRFs with Applications to Key Management

Stanisław Jarecki[1], Hugo Krawczyk[2], and Jason Resch[3]

[1] University of California, Irvine
[2] IBM Research
[3] IBM

Abstract. An Oblivious PRF (OPRF) is a protocol between a server holding a key to a PRF and a user holding an input. At the end of the interaction, the user learns the output of the OPRF on its input and nothing else. The server learns nothing, including nothing about the user's input or the function's output. OPRFs have found many applications in multiple areas of cryptography. Everspaugh et al. (Usenix 2015) introduced Partially Oblivious PRF (pOPRF) in which the OPRF accepts an additional non-secret input that can be chosen by the server itself, and showed applications in the setting of password hardening protocols. We further investigate pOPRFs showing new constructions, including distributed multi-server schemes, and new applications. We build simple pOPRFs from regular OPRFs, in particular obtaining very efficient DH-based pOPRFs, and provide (n, t)-threshold implementation of such schemes.

We apply these schemes to build Oblivious Key Management Systems (KMS) as a much more secure alternative to traditional wrapping-based KMS. The new system hides keys and object identifiers from the KMS, offers unconditional security for key transport, enables forward security, provides key verifiability, reduces storage, and more. Further, we show how to provide all these features in a distributed threshold implementation that additionally protects the service against server compromise. Finally, we extend the scheme to a threshold Oblivious KMS with *updatable encryption* so that upon the periodic change of OPRF keys by the server, an efficient update procedure allows a client of the KMS service to non-interactively update all its encrypted data to be decryptable only by the new key. Our techniques improve on the efficiency and security of several recent works on updatable encryption from Crypto and Eurocrypt.

We report on an implementation of the above schemes and their performance, showing their practicality and readiness for use in real-world systems. In particular, our pOPRF constructions achieve speeds of over an order of magnitude relative to previous pOPRF schemes.

1 Introduction

Oblivious Pseudorandom Functions (OPRF) [44,23] are interactive schemes between a server holding a key to a PRF and a user holding an input. At the

end of the interaction the user learns the output of the PRF on its input and the server learns nothing (neither the input nor the output of the function). OPRFs have found numerous applications including private set intersection [29,1,23,27,37,19], password protocols [22,34,36], searchable encryption [23,14,33], file de-duplication [6], pseudonymization [12], and more, and have served as a basis for other primitives such as Private Information Retrieval and Oblivious Transfer. Very efficient implementations of OPRFs are known, particularly those based on the Diffie-Hellman (DH) problem in regular elliptic curve groups [16,47,43,29,22] (see Fig. 1).

In this work we are concerned with OPRF applications where a server provides an "OPRF service" to multiple clients, allowing each client to compute OPRF values under a client-dedicated OPRF key, with the server learning neither the queries nor the responses. Everspaugh et al. [20] observed that in many OPRF-as-a-service applications it is essential for the server to be able to provide its own input to the OPRF computation, e.g., entering the identity of clients to enforce domain separation or to enact rate limits on OPRF responses. An OPRF extension which allows for such server-side input was formalized in [20] as a *Partially Oblivious PRF (pOPRF)*.

The work of [20] considered two further important features desirable from pOPRFs: Verifiability and Updatability. Verifiability refers to the ability of a client receiving an output from the OPRF service to verify that the function was computed correctly. Updatability refers to a setting where OPRF keys are periodically refreshed, thus necessitating updates to data stored on the client side that was generated using the OPRF. In this context [20] put forth the notion of *updatable (p)OPRF* where upon the change of the (p)OPRF key, the client can obtain from the server a short piece of information which it can use to update all the relevant data on the client side without further server interaction.

Everspaugh et al. [20] put all these notions to work in a system, called Pythia, designed as a password hardening service to aid against offline dictionary attacks on stolen password hashes, where the pOPRF is implemented using bilinear groups and pairings. Pythia also presents an architecture that serves multiple clients and where individual client pOPRF keys are derived from a common master key via a regular PRF f (e.g., AES, HMAC, etc.). That is, pOPRF keys are computed as $f_K(i)$ where K is a master key and i and identifier for the OPRF key, e.g. a client ID, version number, secret value, etc. This approach is particularly beneficial when the master key K and the pOPRF operation are protected inside a secure enclave (HSM, SGX, etc.). Using random independent pOPRF keys would require storing such keys outside the often-limited secure enclave storage, reducing security and adding to the system cost. At the same time, such an architecture can still support periodic updates of individual pOPRF keys by varying the identifier i used to derive the OPRF key. While such identifiers may require external storage, these are typically less sensitive than the OPRF keys themselves.

Note, however, that such master key K for PRF f becomes a very valuable secret and an attractive target for attack. A natural defense (ensuring both

availability and secrecy) is to protect the master key via a (n, t)-*threshold scheme* where the pOPRF service is distributed across n servers such that $t + 1$ of these need to cooperate to compute the function while an attacker compromising t of the servers can do nothing to subvert the service or learn its keys.

Our Contributions

In this paper we investigate new constructions of pOPRFs and *Threshold* pOPRFs, showing their remarkable efficiency (particularly when compared to [20]), and we present a range of novel applications of (Threshold) pOPRFs geared to building more secure and reliable key management systems.

Simple pOPRF. While [20] offers a flexible and highly-functional pOPRF, its use of bilinear groups and costly pairings and target-group exponentiations makes it significantly less efficient than the known DH-based OPRF schemes (cf., Fig. 1) that require only three regular elliptic curve exponentiations: one for the server and two for the client. We show that in many applications it suffices to use a simple pOPRF construction whose cost matches the cost of an OPRF. Namely, we define a pOPRF F' as $F'_K(x, y) = F_{f_K(y)}(x)$, where F is any OPRF and f_k a regular PRF. Here, x is the oblivious input and y the non-oblivious one. We note that the fact that each input y defines a different key for the OPRF F limits the applicability of the scheme in some settings, e.g., it cannot simultaneously satisfy the requirements of fine-grain rate limiting and efficient updatability as in [20]. But whenever it works (as is the case in our own applications), one obtains a far more efficient scheme than Pythia. We will refer to this generic construction as *Simple pOPRF (SpOPRF)*.

Threshold pOPRF. While very efficient (when implemented with a DH-based OPRF), the derivation of OPRF keys via a PRF raises difficulties in implementing this scheme as a threshold system. Fortunately, we are able to show a practical implementation of a *Threshold pOPRF* using the above generic SpOPRF approach and the *share conversion* technique of Cramer, Damgard and Ishai [18]. Combining these techniques with recent results on Threshold OPRFs [36], we present a threshold implementation of a DH-based SpOPRF with remarkable performance for a moderate number of servers. To compute the SpOPRF function in a (n, t)-threshold setting, each server performs $\binom{n-1}{t}$ regular-PRF operations (AES encryptions) and a *single* exponentiation, while the client performs *just two* exponentiations, independently of t or n. (Without the share conversion technique this scheme would cost $\binom{n-1}{t}$ exponentiations by the server, making it impractical.) We show that in spite of the exponential nature of $\binom{n-1}{t}$, our threshold pOPRF scheme is highly efficient for moderate but realistic values of (n, t).

Verifiable Threshold pOPRF. We allow clients to verify the pOPRF server operations (as needed in our key management application to prevent loss of data due to incorrect keys), through an interactive procedure which essentially adds one OPRF exchange between server(s) and client that runs in parallel to

the regular OPRF instance, hence preserving two important properties of our constructions (in the common case that verification succeeds): a single-round and a constant number of exponentiations for each server and client, that is independent of the threshold parameters n, t. Only if verification fails in the threshold case, will the client do work proportional to the number of servers that provided shares to the computation.

Oblivious Key Management Service (OKMS). Key Management Systems (KMS) are essential components of storage systems responsible for providing keys for the encryption and decryption of data. Existing KMS schemes (including large cloud-based operations [3,42,30,25]) use the *wrap-unwrap approach* for protecting data encryption keys (dek). Namely, when a client C encrypts a data object under a dek key, it sends dek to the KMS who wraps (i.e., encrypts) dek under a KMS key and returns the result (wrap) to C who stores wrap. When C needs to retrieve dek for decryption, it sends wrap back to KMS who unwraps (i.e., decrypts) it and sends dek back to C. In this approach dek is exposed to the KMS and to attackers on the channel between KMS and client. Moreover, dek is visible to any middlebox and endpoint where TLS traffic is decrypted.

Here we propose to greatly improve KMS security by resorting to a service, called an *Oblivious KMS (OKMS)*, that uses a pOPRF run between clients and the KMS to derive dek keys. The client, in interaction with the KMS, inputs an object identifier to the pOPRF and uses the output as a dek (either for encrypting or decrypting that object). Thanks to the pOPRF properties, the KMS *never learns* dek or even the object for which the key is computed. Moreover, the system does not rely on an external secure channel (e.g., TLS) to transport dek; instead dek is protected by the very nature of pOPRF (and with *unconditional security* in the case of our DH-based implementation of pOPRF).

We further enhance our OKMS design by adding other features not available to traditional KMS schemes: Verifiability (to prevent data loss upon provisioning of incorrect keys), forward security when using unpredictable object identifiers, self-authorization via oblivious inputs, and the strong security against server compromise afforded by a threshold implementation. Moreover, when OKMS derives per-client pOPRF keys via a single PRF, server-side storage costs for storing keys is greatly reduced. Client-side storage is also reduced as the client needs to keep only object identifiers and not key wraps.

Updatable OKMS. As an important and non-trivial extension of the above OKMS design, we present an Updatable Oblivious KMS (UOKMS) that adds to the OKMS the feature of *updatable encryption* [8]. As in the case of OKMS, UOKMS provisions clients with data encryption keys derived via a pOPRF service (keyed with a client-dedicated pOPRF key). However, in an OKMS system a periodic update of a client's pOPRF key would require decryption and re-encryption of *all* the data stored at the client (with each operations requiring a separate pOPRF call to the server to retrieve the corresponding dek). By contrast, an UOKMS system allows for far more efficient transitioning to a new key: Upon change of a client's pOPRF key, the KMS server sends to the client a single *short* update token Δ, computed as a function of the old and new pOPRF keys.

The client can then use Δ to locally update its storage so that its data becomes decryptable by the new pOPRF key but not by the old one, thus providing *forward security*. The system is very efficient in that the update is performed without resorting to further interaction with the UOKMS and without decrypting and re-encrypting any data.

We implement our UOKMS system using a malleable variant of pOPRF that allows for updatability[4], and hence for forward security. In our solution a client keeps a value called a *wrap* for each data object, from which the UOKMS server derives the corresponding data-encryption key dek. Updates are implemented by only changing these wraps. Our updatable KMS solution is more efficient than all recently proposed updatable encryption schemes from Crypto and Eurocrypt [8,9,21,40], and it is the first to support an *oblivious* KMS server. Moreover, ours is the first treatment of updatable KMS with security in the presence of arbitrary decryption and update queries.[5] Most importantly, we are able to provide this highly-secure solution with *remarkable performance* and full compatibility with our *threshold pOPRF* scheme.

Formal analysis. As further contributions we provide definitions for pOPRF (in a stronger sense than [20]) using the UC definitions of OPRF and threshold OPRF from [35,36] and prove the security of our threshold pOPRF scheme in this setting. Additionally, we introduce a model of oblivious updatable encryption in which we analyze our updatable oblivious KMS. The latter model improves on existing definitions of updatable encryption (or encryption with *key rotation*) [8,9,21,40], most importantly by (1) allowing adversary unfettered access to both decryption and update oracles, thus defining a CCA-like notion of updatable encryption, and by (2) supporting interactive decryption protocols which in particular can be *oblivious* to the KMS server.

Performance. In Section 7 we present detailed performance information from our implementation of both OKMS and UOKMS solutions showing the practicality of our techniques. For the OKMS threshold setting, we show the ability of servers to support a large number of clients per second (with total times for serving a client under 1 millisecond) for practical values of $(n, t+1)$. Examples of such pairs include $n \leq 13$ with any $t < n/2$, $(15, 5)$, $(20, 4)$ and any $n \leq 40$ with reconstruction threshold of 3. Client performance time is approximately 0.4 msec for a wrap and 0.2 msec for an unwrap. For the UOKMS system, performance is even better: a client can sustain over 41000/6000/14000 for wrap/unwrap/update operations per second respectively, with a single CPU core, and server operations are only needed for unwrapping. As a point of comparison, a *single* pairing in the Pythia pOPRF computation [20] takes 1 msec, making our implementation over an order of magnitude faster.

[4] Technically, this variant dispenses with the outer hash in the OPRF instantiation from Fig. 1. This results in a weaker primitive equivalent to what [20] defines as a pOPRF, but which we show to be sufficient for the UOKMS system.

[5] Everspaugh et al. [21] define CCA-like security for *ciphertext-dependent* updatable encryption, where the KMS needs to compute a separate update token Δ for each updated ciphertext.

We conclude with impressive throughput and latency results from a prototype implementation of the (U)OKMS Server deployed to an Amazon EC2 instance. We find this implementation capable of answering over 30,000 requests per second with a single server node.

Organization. Section II presents our designs for Partially Obivious PRFs and their threshold extensions. Section III provides the formal foundation to our analysis with UC definitions of secure pOPRFs and Threshold pOPRFs. Section IV discusses verifiability techniques. Section V describes our OKMS solution (and other threshold pOPRF applications) while Section VI presents the UOKMS scheme and its formal analysis. Section VII describes our implementation and performance results.

2 Building a Threshold Partially Oblivious PRF

In this section we present our main Threshold Partially OPRF construction based on Diffie-Hellman that can accommodate very efficient elliptic curve groups. It adapts the Threshold OPRF function from [36] to the partially OPRF setting following the generic "Simple pOPRF" approach discussed in the introduction. The practicality of the function uses in an essential way the share conversion technique from Cramer, Damgard and Ishai [18].

Notation and basic components. $[n]$ denotes the set $\{1, \ldots, n\}$; "$\leftarrow$" denotes a randomized assignment; "$\leftarrow_{\text{R}}$" denotes uniform sampling from a given set. Across the paper we use the following functions. H, H' are hash functions (modeled as random oracles) with arbitrary inputs and with G and $\{0,1\}^{\ell}$, respectively, as their ranges. Here, G is a cyclic group of prime order q and ℓ is a security parameter ($|q| \geq 2\ell$). Functions f, f' denote regular pseudorandom families, e.g., those built on top of AES or HMAC (sometimes with the range reduced modulo q). We define secret sharing schemes with parameters (n, t), implemented using Shamir's scheme with polynomials of degree t over $\mathbb{Z}_q$. Thus, the scheme requires the cooperation of $t+1$ parties to reconstruct a secret or compute a threshold function. For generality, we denote by $\alpha_1, \ldots, \alpha_n$ the evaluation points of such a scheme (one often uses $\alpha_i = i$).

2.1 DH-based OPRF Schemes

We first recall the DH-based OPRF constructions from [35,36] that we use as a basis for our Partially Oblivious PRF (pOPRF) schemes. Figure 1 shows the DH-based OPRF construction while Figure 2 shows a Threshold version of the same function. The functions were called 2HashDH in the above papers but here we rename them to dh-op (for DH-based OPrf) and tdh-op ('T' for threshold), respectively. We note that tdh-op is described in a simplified form in Figure 2 where the set of reconstruction parties $\mathcal{SE}$ is assumed to be known by U in advance. If the reconstruction set $\mathcal{SE}$ is not known a-priori (i.e., more than $t+1$ servers are contacted), each S_i would respond with a^{k_i} and U would compute the

interpolation in the exponent at the cost of a single multi-exponentiation (which can be further optimized when the α_i's are small, e.g., $\alpha_i = i$, using a recent technique from [45]). An additional important feature of the tdh-op solution is that the aggregation of server values b_i into the dh-op result can be done by a proxy server (one of the threshold servers or a special purpose one) so that the threshold implementation is transparent to the user. The schemes we present inherit this feature.

PRF F_k Definition
For key $k \leftarrow_R \mathbb{Z}_q$ and $x \in \{0,1\}^*$, define

$$F_k(x) = H(x, (H'(x))^k)$$

Oblivious F_k Evaluation between user U and server S

1. On input x, U picks $r \leftarrow_R \mathbb{Z}_q$; sends $a = (H'(x))^r$ to S.
2. S verifies that the received a is in group G and if so it responds with $b = a^k$.
3. U outputs $F_k(x) = H(x, b^{1/r})$.

Fig. 1. DH-based OPRF function dh-op

Key and server initialization.
Random key $k \leftarrow_R \mathbb{Z}_q$ is secret shared using Shamir's scheme with parameters n, t; Each server S_i, $i \in [n]$, receives share k_i.

Threshold Oblivious Computation of $F_k(x)$.

- On input x, user U picks $r \leftarrow_R \mathbb{Z}_q$ and computes $a := H'(x)^r$; it chooses a subset $\mathcal{SE}$ of $[n]$ of size $t+1$ and sends to each server $S_i, i \in \mathcal{SE}$, the value a and the subset $\mathcal{SE}$.
- Upon receiving a from U, server S_i verifies that $a \in G$ and if so it responds with $b_i := a^{\lambda_i \cdot k_i}$ where λ_i is a Lagrange interpolation coefficient for index i and index set $\mathcal{SE}$.
- When U receives b_i from each server $S_i, i \in \mathcal{SE}$, U outputs as the result of $F_k(x)$ the value $H(x, (\prod_{i \in \mathcal{SE}} b_i)^{1/r}))$.

Fig. 2. Protocol tdh-op [36]: (n,t)-threshold computation of dh-op from Fig. 1

2.2 Threshold Partially Oblivious PRF

Recall the "Simple pOPRF" (SpOPRF) generic scheme described in the introduction that builds a pOPRF scheme F' from any OPRF F and regular PRF f, namely,

$$F'_K(x, y) = F_{f_K(y)}(x) \tag{1}$$

where x and y are the oblivious and non-oblivious inputs to F', respectively. When using dh-op (Fig. 1) as the OPRF F and a regular PRF f with range in $\mathbb{Z}_q$ one gets the following DH-based SpOPRF that we denote by dh-pop^f:

$$\mathsf{dh\text{-}pop}^f_K(x, y) = H(x, H'(x)^{f_K(y)}) \tag{2}$$

Our goal is to provide a threshold implementation of this function. The idea is to define a PRF f for which each threshold server can generate, independently and for every value of y, linear shares of $f_K(y)$. These can then be used in lieu of the shares k_i in the threshold OPRF protocol tdh-op from Fig. 2. We obtain such a scheme following the work of Cramer, Damgard and Ishai [18] in two stages. First, in Fig. 3, we present an "intermediate" single-server pOPRF scheme dh-pop$^{\mathcal{F}}$ that realizes the SpOPRF function from (2) with a special PRF family $\mathcal{F}$. Due to our intensive use of this function we will often omit the $\mathcal{F}$ superscript when it's clear from the context. The family $\mathcal{F}$ originates from the work of Naor et al. [43] (based on the replicated secret sharing technique of Ito et al. [32]). As shown in [43] this scheme can already be distributed as a threshold scheme between n servers where each server $S_i, i \in [n]$, is given the set of $\binom{n-1}{t}$ keys $\{k_A : A \in \mathcal{A}_i(n, t)\}$. Indeed, as it is easy to verify, no coalition of t servers can reconstruct the full collection of keys $\mathcal{K}$, hence no such coalition can compute the sum $K = \sum_{k_A \in \mathcal{K}} f'_{k_A}(y)$ and obtain $\mathsf{dh\text{-}pop}_{\mathcal{K}}(x, y)$. However, computing such scheme would cost $\binom{n-1}{t}$ exponentiations per each server. Instead, we transform $\mathsf{dh\text{-}pop}_{\mathcal{K}}(x, y)$ into a much more efficient scheme by applying to it the *share conversion* technique from [18] (recalled below). Then, combining the resultant protocol with the Threshold OPRF scheme of Fig. 2 we obtain *our main DH-based Threshold pOPRF scheme,* tdh-pop*,* presented in Fig. 4.

We recall the share conversion technique next, followed by a lemma that states the correctness of the tdh-pop construction.

Share conversion [18]. Let $\alpha_1, \cdots, \alpha_n$ denote n elements in $\mathbb{Z}_q$. For any subset A of $[n]$ of size t, define $p_A(x)$ as the unique polynomial of degree t over $\mathbb{Z}_q$ such that $p_A(0) = 1$ and $p_A(\alpha_i) = 0$ for $i \in A$. For $i \in [n]$ and $A \in \mathcal{A}_i$ (i.e., any subset of $[n]$ of size t that does *not* contain i), denote $p_{A,i} = p_A(\alpha_i)$. It is easy to see that $p_A(x) = \prod_{j \in A}(1 - \frac{x}{\alpha_j})$ and therefore $p_{A,i} = \prod_{j \in A}(1 - \frac{\alpha_i}{\alpha_j})$.

Lemma 1.*[18] Let f be a pseudorandom function and $\mathcal{K} = \{k_A : A \in \mathcal{A}(n, t)\}$ a collection of sets as defined in Figure 3. Given y in the domain of f, the values $k_i(y), i \in [n]$ defined as*

$$k_i(y) = \sum_{A \in \mathcal{A}_i} p_{A,i} \cdot f'_{k_A}(y) \tag{3}$$

Set collections. Given integers n, t, $0 \leq t < n$, let $\mathcal{A}(n,t)$ denote the collection of subsets of $[n]$ of size t and let $\mathcal{A}_i(n,t)$, for $i \in [n]$, denote the collection $\{A \in \mathcal{A}(n,t) : i \notin A\}$ (we often omit the explicit parameters n, t when referring to $\mathcal{A}$ and $\mathcal{A}_i$).

Key collection. Let f' be a PRF family with outputs in $\mathbb{Z}_q$ and $\mathcal{K}$ be a set of $\binom{n}{t}$ values $\{k_A : A \in \mathcal{A}\}$ where each k_A is chosen at random from the space of keys of PRF f'.

pOPRF function dh-pop. Given a PRF f' and key collection $\mathcal{K}$ as above, we define a PRF family $\mathcal{F}_{\mathcal{K}}(y) = \sum_{k_A \in \mathcal{K}} f'_{k_A}(y)$ on the same domain as f'. We then use $\mathcal{F}$ as the function f in the SpOPRF scheme (2) to define dh-pop:

$$\mathsf{dh\text{-}pop}_{\mathcal{K}}(x,y) = H(x, H'(x)^{\sum_{k_A \in \mathcal{K}} f'_{k_A}(y)})$$

Note: $\mathsf{dh\text{-}pop}_{\mathcal{K}}(x,y) = \mathsf{dh\text{-}op}_K(x)$ for $K = \sum_{k_A \in \mathcal{K}} f'_{k_A}(y)$)

Fig. 3. DH-based Partially Oblivious PRF dh-pop (also denoted $\mathsf{dh\text{-}pop}^{\mathcal{F}}$)

form a (n,t)-secret sharing of $K = \sum_{k_A \in \mathcal{K}} f'_{k_A}(y)$ (i.e., all $k_i(y)$ lie on the same polynomial of degree t whose free coefficient is K).

Performance. tdh-pop preserves the minimal number of exponentiations enjoyed by tdh-op (Fig. 2); indeed the only additional computation is the calculation (3). While the latter has complexity $\binom{n-1}{t}$, we show in Section 7 that for typical practical values of n, t, our schemes remain remarkably efficient. In applications where servers reuse often the key $k_i(y)$, its value can be computed once and cached, avoiding the cost of (3). Finally, we remark that the user-side tdh-pop computation is the same as for the fully oblivious tdh-op function hence the same client code works in both cases. See Sec. 7 for more performance information.

Security of dh-pop and tdh-pop. The protocols are proven in the random oracle model assuming the Gap One-More DH assumption (and its stronger version from [36]) and the pseudorandomness of f'. See Section 3.

Adding verification. Supporting verification of the server's actions is crucial for some applications. For example when using a (partially) OPRF to derive data encryption keys, a wrong OPRF computation would produce an irrecoverable encryption key which leads to irrecoverable loss of data. We present verification techniques in Section 4.

Other OPRF variants. The above techniques for designing pOPRFs and Threshold pOPRFs can apply to other schemes. First, we note that the BLS scheme of Boneh et al. [10] can be seen as an implementation of the dh-op scheme over a pairings-friendly group. While this results in a less efficient scheme (and a more restrictive choice of groups), it provides a natural way of verifying the correct output from the OPRF using the BLS signature itself (at the cost of pairing

Key and server initialization. Given parameters n, t, $0 \leq t < n$, assume key collection $\mathcal{K} = \{k_A : A \in \mathcal{A}(n,t)\}$ is chosen as in Fig. 3. Each server $S_i, i \in [n]$, is given the set of $\binom{n-1}{t}$ keys $\{k_A : A \in \mathcal{A}_i(n,t)\}$ as its share.

In addition, for each $i \in [n]$, server S_i precomputes and stores values $p_{A,i} = \prod_{j \in A}(1 - \frac{\alpha_i}{\alpha_j})$ for all $A \in \mathcal{A}_i(n,t)$.

Threshold Evaluation of tdh-pop at Server S_i on non-oblivious input y. S_i interacts with user U (holding an oblivious input x) according to the evaluation procedure of tdh-op as described in Figure 2 where S_i's key k_i is defined as $k_i = k_i(y)$ as in Equation 3, namely,

$$k_i(y) = \sum_{A \in \mathcal{A}_i} p_{A,i} \cdot f'_{k_A}(y).$$

Caching $k_i(y)$. Key share $k_i = k_i(y)$ can be cached for use with multiple oblivious inputs x from U.

Verification. In applications where the user verifies the server's actions (with respect to value y), any of the verification schemes from Section 4 can be applied with $v_i = g^{k_i(y)}$ as the verification key for server S_i.

Fig. 4. Threshold Partially Oblivious PRF tdh-pop

operations at the user) – see Appendix A. Second, the function of Everspaugh et al. [20], defined as $e(H_1(z), H_2(x))^k$ where e is a bilinear map $e : G_1 \times G_2 \to G_T$, and H_1, H_2 are hash functions, can replace dh-op in the previous constructions of this section, thus allowing for a threshold implementation when the keys k to this function are derived from a common master key via a PRF (as done in [20]). While the resultant scheme is computationally costly (it requires pairings and exponentiation in G_T), it allows for a non-oblivious input in addition to the input y in our constructions that can benefit some applications (see Appendix B).

3 Model and Analysis of Threshold Partially Oblivious PRF

3.1 Definitions

Jarecki et al. defined the notion of Oblivious PRF (OPRF) in the Universally Composable (UC) model in [35] and extended it to the Threshold setting (T-OPRF) in [36]. Here we adapt these functionalities to the "partially oblivious" setting where the function receives, in addition to the oblivious input by the user, a non-secret arbitrary input y that we model as an adversary-provided input. We model the pOPRF as a collection of OPRFs where each OPRF is defined by a value y in a domain set Y. Formally, pOPRF is defined via a functionality $\mathcal{F}_{\text{pOPRF}}$ that is identical to functionality $\mathcal{F}_{\text{OPRF}}$ from [35] except for the inclusion

of the value y, which can be chosen arbitrarily by the adversary. This value is used internally by the functionality as a parameter that determines the random table that defines the outputs of the function on input y and oblivious input x. We present the functionality $\mathcal{F}_{\text{pOPRF}}$ in Fig. 5.

The case of a Threshold pOPRF is treated similarly by augmenting the T-OPRF functionality $\mathcal{F}_{\text{TOPRF}}$ from [36] with the additional parameter y that, as before, is chosen by the adversary and fed into the functionality as a parameter for determining the random table from which results of the Threshold pOPRF are taken. We refer the reader to [36] for the details of the T-OPRF functionality. Our extension, $\mathcal{F}_{\text{TpOPRF}}$, is analogous to the one for the single-server case in Fig. 5 and is omitted here.

All tables and tickets are initially undefined.
$p(x)$ is a polynomial that defines the length of the pOPRF output.

- Upon receiving (EVAL, sid, S, x) from user U (resp. $\mathcal{A}^*$), send (EVAL, sid, U, S) (resp. (EVAL, $sid, \mathcal{A}^*, S$)) to $\mathcal{A}^*$, receive back value y; record $\langle sid, U, S, x, y\rangle$ (resp. $\langle sid, \mathcal{A}^*, S, x, y\rangle$).
- Upon receiving (SNDRCOMPLETE, sid, y, S) from $\mathcal{A}^*$, increment $\mathsf{tx}(sid, y, S)$ (or set to 1 if previously undefined) and send (SNDRCOMPLETE, sid, y) to S or to $\mathcal{A}^*$ if S is not a sender's identity.
- Upon receiving (RCVCOMPLETE, sid, P, S, S^*, y) from $\mathcal{A}^*$, recover record $\langle sid, P, S, x, y\rangle$; abort if such tuple does not exist or S is honest and $S^* \neq S$ or $\mathsf{tx}(sid, y, S^*)$ is 0 or undefined. Otherwise decrement $\mathsf{tx}(sid, y, S^*)$ and:
  - If $T(sid, S^*, x, y)$ is defined, then send (EVAL, $sid, T(sid, S^*, x, y)$) to P.
  - Otherwise pick ρ at random from $\{0,1\}^{p(\ell)}$, set $T(sid, S^*, x, y) := \rho$ and send (EVAL, sid, y, ρ) to P.

Fig. 5. Partially-Oblivious PRF Functionality $\mathcal{F}_{\text{pOPRF}}$

3.2 Security of tdh-pop

We first state the pOPRF security of the generic Simple pOPRF scheme and the DH-based derivatives.

Lemma 2. *Let F and f be OPRF and PRF families respectively, then the SpOPRF scheme (1), $F'_K(x, y) = F_{f_K(y)}$, is a pOPRF family, namely, it instantiates functionality $\mathcal{F}_{\text{pOPRF}}$. In particular this applies to the* $\mathsf{dh\text{-}pop}^f$ *scheme of equation (2) and* $\mathsf{dh\text{-}pop}^{\mathcal{F}}$ *as described in Fig. 3.*

proof Immediate from the pseudorandomness and (computational) independence of the resulting OPRF keys $f_K(y)$ for different y's (formally, a hybrid composition of OPRF with a PRF f).

The next theorem derives from the proof [36] showing that the scheme tdh-op is a T-OPRF under the Gap One-More Diffie-Hellman assumption (see Section 6.1 where a similar assumption is presented and replace the inversion oracle with a DDH oracle).

Theorem 1. *The* tdh-pop *protocol from Figure 4 is a Threshold pOPRF, namely, it instantiates functionality* $\mathcal{F}_{\text{TpOPRF}}$*, in the RO model under the Gap One-More Diffie-Hellman assumption.*

proof We need to show that tdh-pop instantiates functionality $\mathcal{F}_{\text{TpOPRF}}$ which boils down to prove that for fixed y, tdh-pop is a Threshold OPRF according to [36]. Namely, tdh-pop instantiates the functionality $\mathcal{F}_{\text{TOPRF}}$ from that paper. We first note that by fixing y, tdh-pop induces a sharing $(k_1(y), \ldots, k_n(y))$ of the key $K = \sum_{k_A \in \mathcal{K}} f'_{k_A}(y)$. The rest is an identical computation of tdh-op proven a secure Threshold OPRF in [36]. Thus, for fixed y, tdh-pop inherits the OPRF property from tdh-op, except that in tdh-pop the attacker's view also includes the set of keys $\{k_A : A \in \mathcal{A}_i(n,t)\}$ for each compromised server S_i, information that does not exist in tdh-op. If we can prove that a tdh-op attacker can simulate these sets of keys given only the shares of corrupted parties in tdh-op we are done. We show this next.

To facilitate the proof, we consider a modified tdh-pop scheme where the values $f'_{k_A}(y)$ are replaced with truly random values r_A. Thus we need to prove that the attacker (denoted Adv) against tdh-op can simulate these values given the shares of compromised parties. (Note that by showing security of tdh-pop under this modification, we imply security for tdh-pop for any PRF f' – indeed, a successful attacker against tdh-pop with PRF f' would imply a distinguishing (from random) attack against f). For simplicity, we also assume Adv has corrupted parties $S_1, \ldots, S_t$. The case of less than t corruptions and of different identities is analogous.

We show how Adv generates the values r_A for all $A \in \mathcal{A}(n,t)$ using the following linear system (over $\mathbb{Z}_q$) induced by equation (3) and by the values $p_{A,i} = p_A(\alpha_i)$ where $p_A(x)$ is the unique polynomial of degree t over $\mathbb{Z}_q$ with free coefficient 1 and roots $\alpha_j, j \in A$. The system is defined as $P \cdot \bar{r} = \bar{s}$ where P is a $t \times m$ matrix with $m = \binom{n}{t}$; row i corresponds to party S_i; each column ℓ corresponds to a set $A_\ell \in \mathcal{A}(n,t)$ (in some fixed enumeration of sets in $\mathcal{A}(n,t)$), and $P_{i,\ell}$ is defined as $p_{A_\ell}(\alpha_i)$. The vector $\bar{s}$ has share s_i of party $S_i, i \in [t]$ in its i-th entry. Finally, the m-coordinate vector $\bar{r}$ represents the unknown in the linear system and it is used to determine the values $\{r_A : A \in \mathcal{A}(n,t)\}$. Indeed, note that the solutions to the above system correspond to the sets of values r_A that result in shares $s_1, \ldots, s_t$ for parties $S_1, \ldots, S_t$. Thus, if Adv can sample a random solution in this set we obtain the simulated view of a tdh-pop attacker as required.

What remains to show is that this linear system has a solution for any $s_1, \ldots, s_t$ or, equivalently, that matrix P has rank t. For each $i \in [t]$, consider

a set A_{ℓ_i} that contains all values in $[t]$ except i. Note that the columns of P corresponding to A_{ℓ_i} has zeros in all rows $j, j \in A_{\ell_i}$, and a non-zero value in the i-th row (since by definition $p_{A_{\ell_i}}(\alpha_j) = 0$ if and only if $j \in A_{\ell_i}$). Thus, the submatrix corresponding to the selected t sets A_{ℓ_i} is diagonal, hence the rank of P is t.

4 Verifiable Partially Oblivious PRF

As mentioned in Sections 1 and 2, supporting verification of the server's actions is crucial for many applications. For example, when a (partially) Oblivious PRF is used to derive data encryption keys, a wrong OPRF computation would produce an irrecoverable encryption key and lead to irrecoverable loss of data. Here we describe mechanisms for implementing such verification for the pOPRF and OPRF schemes from Sec. 2, including their threshold versions.

In the case of the dh-op scheme we assume that the client C has the (certified) value $v = g^k$ where g is a generator of the group G and k is the server's OPRF key. We refer to v as the server's OPRF verification key. For SpOPRF scheme like dh-pop^f (equation 2), the verification key becomes $g^{f_k(y)}$. In the threshold case, to verify the scheme tdh-pop (Fig. 4), the client would need the values $v_i = g^{k_i}$ for each participant server S_i as well as g^k where k is the value shared through $k_i, i \in [n]$ ($v = g^k$ can be computed from the v_i values through interpolation in the exponent).

There are several methods we can adopt for OPRF verification. Below we describe an *interactive* procedure which we adopt for our implementation due to its simplicity and performance. In Appendix A we recall alternative *non-interactive* mechanisms for verifying exponentiation correctness.

The solution presented here works by running OPRF on two different values and then verifying consistency via a single multi-exponentiation by the client. This greatly simplifies the verification implementation as it is essentially a repeat of the operations in the basic schemes (this holds for the single-server and threshold cases). The added computational cost is a single exponentiation for the server(s) and two multi-exponentiations for the client, essentially doubling the work for the non-verified case. It maintains the property that in normal operation (in which verification succeeds) the number of exponentiations is independent of the parameters n, t in the system. In the threshold setting, only if verification fails will the client test (without further interaction) the individual contributions of the different threshold servers.

We first describe the scheme for the case of the single-server OPRF dh-op from Fig. 1. The procedure is reminiscent of Chaum's protocol for undeniable signatures [15] but simplified by dispensing of zero-knowledge proofs that are not needed here. The integrity guarantee is unconditional, namely against unbounded attackers.

- On input x, C sets $h = H'(x)$, sets $r, c, d \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, and sends to S the pair of values $a = h^r$, $b = h^c g^d$.

– S responds with $A = a^k, B = b^k$.
– C checks that

$$A^{r'} = B^{c'} v^{-dc'} \tag{4}$$

where $r' = r^{-1}, c' = c^{-1}$. It rejects if the equality does not hold, otherwise C sets the value of $(H'(x))^k$ to $A^{r'}$ which it already computed for equation (4).

Lemma 3. *(1) If $A \neq a^k$ or $B \neq b^k$, C accepts with probability at most $1/q$. (2) Security of the OPRF is preserved in spite of the additional verification query.*

proof The proof of (1) is standard: We write equation (4) as powers of g where we denote $A = a^{k_1}, B = b^{k_2}, h = g^{\ell}$. By equating the exponents in this expression we get $\ell(k_1 - k_2) = \ell + e(k_2 - k_1)$ where $e = d/c$. This equation has one solution at $k_1 = k_2 = k$ (the honest case), otherwise we have that for each value of k_2 there is a single value of k_1 that satisfies the equation, namely, $k_1 = e(k_2 - k)/\ell + k_2$. However, since e is uniformly distributed over Z_q (since b hides the value of c perfectly) then the probability to find such pair (k_1, k_2) is $1/q$ even for an unbounded attacker.
Claim (2) follows from the fact that the only added computation relative to dh-op is one more query to the oblivious function adding to the number of queries but preserving the OPRF security.

Note on Security and Rate-Limitation. The double OPRF operation incurred by the verification step needs to be considered in settings where the server imposes rate limits on OPRF invocations. A corrupted client can abuse the verification calls to obtain OPRF computations on *two* arbitrary inputs of the client's choice, but in typical applications of OPRF the server limits the rate of OPRF invocations, so using interactive verification means that a corrupted client can compute OPRF values at the at most twice faster rate.

Interactive Verification in Threshold OPRF protocols. We now adapt the scheme to the threshold functions tdh-op and tdh-pop. The client C sends the same pair of values (a, b) to each participant server S_i who responds with $A_i = a^{k_i}, B_i = b^{k_i}$. Upon gathering $t + 1$ responses, C interpolates in the exponent (one multi-exponentiation) to obtain values A, B and checks the identity (4). If it holds, C sets $(H'(x))^k$ to $A^{r'}$, else it applies the check (4) to each pair A_i, B_i received by participating server S_i using $v_i = g^{k_i}$ instead of v. The cost of this procedure in the normal case, where the verification against $v = g^k$ succeeds, is the same as the single-server case except for one additional interpolation in the exponent. If verification against $v = g^k$ fails then the cost is an additional multi-exponentiation per each participating server.
Non-Interactive Verification for (Partially) Oblivious PRFs

In Appendix A we recall two standard *non-interactive* verification methods that apply to our Partially Oblivious PRF protocols, including their threshold versions.

Base case: KMS server S stores a separate OPRF key for each of its clients.

Functions: OPRF F and symmetric authenticated encryption scheme Enc.

OPRF Keys: S stores a client-specific OPRF key k_c for each client.

Encryption of object Obj *by client* C*:* C runs OPRF protocol with S, where C inputs object identifier ObjId and S inputs key k_c. C sets $\mathsf{dek} = F_{k_c}(\mathsf{ObjId})$ and stores the pair $(\mathsf{ObjId}, \mathsf{Enc}_{\mathsf{dek}}(\mathsf{Obj}))$

Decryption of encrypted object ObjId *by client* C*:* As in the encryption case, C interacts with S to compute $\mathsf{dek} = F_{k_c}(\mathsf{ObjId})$ and decrypts Obj using dek.

Storage efficient variant: Client-specific OPRF keys k_c are derived as $f_{\mathsf{mk}}(\mathsf{Cid})$ where f is a PRF, mk a server "master key", and Cid a client identifier.

Threshold variant of base case: As above but OPRF replaced with a Threshold OPRF (e.g., $\mathsf{tdh\text{-}op}$ of Fig. 2).

Threshold implementation of storage-efficient variant: Replace PRF f and OPRF F with the $\mathsf{tdh\text{-}pop}$ scheme of Fig. 4.

Verification of correct computation of dek**:** Use the verification procedure corresponding to the (p)OPRF in use (Sec. 4).

Fig. 6. Oblivious KMS

5 Applications to Key Management

We show applications of the techniques from the previous sections to the key management setting. In the next section we will show another such application but in the context of updatable encryption.

5.1 Oblivious Key Management System

Standard key management systems (KMS), e.g., [3,42,30,25], support applications that encrypt data by providing a "key wrapping service". Consider, for example, a storage application that encrypts objects (files, media, etc.) before storing them. When object Obj is to be stored, the application running at a client C chooses a key dek (for "data encryption key") and encrypts Obj using a symmetric cipher under key dek. The application then sends dek to the key management system KMS (together with credentials authenticating C) who encrypts dek under a "client root key" dedicated to client C. This encryption operation applied to the data encryption key dek is known as *wrapping* and the ciphertext encrypting dek is called the *wrap,* which we denote by wrap. KMS then sends back wrap to C which stores it with the encrypted object[6]. When decryption of the same object is needed, the corresponding wrap is sent back to

[6] In some implementations, dek is chosen by KMS itself and sent to the client together with its wrap.

KMS who "unwraps" (i.e., decrypts under C's root key) to obtain dek and send it back to C. Note that the key dek travels from C to KMS (at the wrapping request) and from KMS to C (at the unwrapping request) only protected by the channel between KMS and C. This exposes dek to weaknesses of such channel, e.g., certificate and other man-in-the-middle attacks on TLS, TLS termination outside the safe boundaries of the application (e.g., at a middlebox or a CDN service), and more.

As a powerful application of OPRFs in the KMS setting, we propose to replace the wrapping approach with an OPRF service as shown in Figure 6. When a client C needs to compute a dek for encrypting an object named ObjId, C interacts with a key management server who runs an OPRF F on behalf of the client and together they compute $\mathsf{dek} = F_{k_c}(\mathsf{ObjId})$, where k_c is an OPRF key dedicated to C (a "client root key") and ObjId is the name with which C identifies the object. Thanks to the OPRF properties the value of dek is hidden from the server and from the network (with perfect secrecy in the case of the DH-based implementations from Section 2), dispensing with the need for secure channels and their associated weaknesses. Also the value of ObjId (e.g., a file or document name), which often carries confidential or private information, is hidden from the network and server. Moreover, in the case that ObjId values are chosen by C so that they are unpredictable to an attacker, a strong form of *forward security* is achieved: dek remains secure even if the OPRF key is eventually compromised. In addition, unpredictability of the input to the OPRF acts as a form of "self-authorization" in that only those knowing this input can access the corresponding dek.

We describe the OPRF-based KMS scheme, to which we refer as *Oblivious KMS Protocol*, in Fig. 6. There we present several important variants that include space-efficient schemes and threshold implementations. The space-efficient implementation where client keys are derived from a single PRF can support O(1) memory, namely, support an unbounded number of clients with fixed storage. In some cases, however, in order to allow for key deletions, keys are associated with secret identifiers that require storage [20]. However, such identifiers may use less expensive storage (relative to OPRF keys that are stored in secure enclaves) and their number is often significantly smaller than the number of OPRF keys (e.g., a full ensemble of keys can use the same identifier, say a project name, if these keys are deleted or updated in tandem).

At the same time, any centralized KMS service becomes a target for attack, hence it is important to be able to support it as a distributed service via a threshold implementation. This is indeed another major advantage of the OPRF approach over the traditional one as OPRFs admit very efficient threshold implementations. Indeed, in the base scheme of Fig. 6, where the server stores separate, independent OPRF keys for each of its clients, one can use the very efficient tdh-op scheme (Fig. 2) as the Threshold OPRF solution applied to each individual client key k_c. Such a scheme also enjoys the benefits of an efficient proactive security extension [28]. Thresholdizing the space-efficient variant is more challenging but our tdh-pop method from Fig. 4 addresses this too. Note

that in this case, *all* the distributed servers enjoy the storage efficiency of the scheme.

Finally, to allow the client to verify the correctness of a computed key dek (which if incorrect would imply irrecoverable loss of data), the client will be provisioned for each key k_c with a certified verification key $v_{k_c} = g^{k_c}$. For the threshold case the client is also given the individual verification keys g^{k_i}. Any of the verification methods in Sec. 4 apply here. (Note that verification is essential for the encryption operation; for decryption one can test the key computed by the OPRF against the stored authenticated ciphertext.)

We end by noting that upgrading an existing wrapping-based system to an oblivious KMS as above is greatly simplified as one can adapt existing wrapping APIs to the new functionalities. Also, note that the oblivious KMS dispenses with the need to store wrap values leading to reduced storage on the client side. One still needs object identifiers but these exist independently of encryption (even if these identifiers are randomized, one can support that with a small number of randomization seeds).

In all, we see that the Oblivious KMS scheme enjoys major advantages relative to today's standard approach to KMS, including: (i) it provides a significantly more secure approach than the current wrapping approach in terms of key transport, forward secrecy, and object identity confidentiality; (ii) supports key verification; (iii) supports a threshold solution, including proactive security in the base case; (iv) reduces the secure storage requirements on the KMS side; (v) reduces client-side storage; and (vi) provides for an optional mechanism of "self-authorization".

In Section 6 we strengthen the above OKMS scheme even further with support for updatable encryption and with a detailed security model.

5.2 Distributed Password Manager with Unbounded Capacity

Shirvanian et al. [48] describe a password manager service, called SPHINX, based on OPRFs. The idea is for the user to memorize a single master password pwd that is then mapped via an OPRF into individual randomized keys rwd for each service/account the user has access to. That is, the password that the user registers for such an account is computed as $\mathsf{rwd} = F_k(\mathsf{pwd}, \text{"account@service"})$ where F is an OPRF, k is the OPRF key, and the inputs are concatenated as a single input to the function. When the user needs to login to an account, it first retrieves the corresponding password rwd by communicating with the OPRF. The key k can be held by a user's device such as a smartphone or by an online service. Compromising the device or server does not reveal the user master password pwd or service-specific passwords rwd. Such a compromise does allow to run an offline dictionary attack on the user's master password pwd but verifying a correct guess requires interacting with the corresponding service and account, or comparing against a stolen password file which contains a hash of rwd.

So, while such a solution adds significantly to the security of passwords, it still presents an opportunity for an attacker to run offline dictionary attacks

upon device/server compromise. A major defense in this case is to distribute the security via a multi-server threshold implementation, so that *nothing* is learned about the user's password as long as less than a threshold of servers is compromised (and if such a threshold is compromised, an offline attack, possibly with online interaction with a user's account, is still needed).

In a single-server implementation of such an OPRF service, the server can serve an unbounded number of users by deriving user-specific OPRF keys through a regular PRF f and a server's master key mk (i.e., the OPRF key for user U is $f_{\mathsf{mk}}(U)$, thus implementing an SpOPRF as in expression (1)). The only non-constant memory in this case is related to the management of rate limitation (to prevent attackers from running an online dictionary attack on a user's account or master password) but the memory requirements for this are more relaxed than for storage of keys and requires less space. In particular, it only needs to memorize recent queries to the service and can be implemented using efficient data structures such as Bloom filters.

Our work allows for the first time to preserve such memory efficiency even in the case of a multi-server threshold implementation of the above service. Indeed, this is exactly what our threshold pOPRF scheme $\mathsf{tdh\text{-}pop}$ from Fig. 4 provides and does so very efficiently. Note that verification of the server(s) operation is not needed in this case, especially that the user is not required to remember a verification key (only her master password pwd).

Thanks to the strong defense against offline dictionary attacks provided by such a distributed system, this service can be extended to other uses beyond password authentication, e.g., using rwd as a strong cryptographic key for protecting user secrets (e.g., stored bitcoin wallets, cloud storage decryption keys, confidential credentials, etc.). The storage of secrets and other user's information can be done separately from the pOPRF servers, hence preserving their memory efficiency and reduced system complexity.

In Appendix B we show the applicability of our techniques to a different password hardening setting, namely, the password onion scheme from [20].

6 Updatable Oblivious KMS

We expand on the application of threshold partially oblivious PRF to Oblivious KMS shown in Section 5.1 to show a design for a practical *Updatable* Oblivious KMS, which also supports a threshold KMS implementation. An Updatable KMS is otherwise known as *Updatable Encryption* [8,40] or *Encryption with Key Rotation* [21]. Unlike the storage-efficient variant of the Oblivious KMS of Section 5.1, we will use a traditional wrapping approach of storing the wrap values at the client side together with ciphertext $\mathsf{Enc}_{\mathsf{dek}}(\mathsf{Obj})$, i.e. treating triple $c = (\mathsf{ObjId}, \mathsf{wrap}, \mathsf{Enc}_{\mathsf{dek}}(\mathsf{Obj}))$ as an encryption of plaintext Obj under (the public key corresponding to) the server-held key k_c. An Updatable KMS allows the KMS server to change, a.k.a. *rotate*, key k_c assigned to client C to a fresh key $k_c{}'$, and it allows the client to update *all* its stored wraps so that the updated wraps can be decrypted using $k_c{}'$ to obtain the original dek keys. Note that this

mechanism updates only the short wrap part of the ciphertext, and does not re-encrypt the data. Support for key updates provides *forward security* to clients' OPRF keys, i.e. an attacker who learns k_c will not be able to use it to decrypt wraps that were updated after the change from k_c to $k_c{}'$. This is the same goal as in [8,21,40], but we provide it in a system that supports *oblivious* decryption by the KMS server, and as we show in Section 6.2 it also efficiently supports threshold KMS implementation.

The Updatable OKMS scheme shown in Figure 7 supports updates with a single short value Δ sent from KMS to C and requires *no other interaction between C and KMS*: The client C can locally use Δ to update all its ciphertexts. We note that our update uses 1 exponentiation per ciphertext while the *non-oblivious* updatable encryption schemes [8,21,40] all use 2 exponentiations per ciphertext. Moreover, the Updatable Oblivious KMS scheme in Figure 7 *preserves all the advantages* of the Oblivious KMS solutions from the Section 5, including partial blindness, verifiability, and efficient threshold implementation, except that it trades the added storage of wraps for the efficient updatability feature.

The Usage of Oblivious *Weak* PRF. We point out that in contrast to the OKMS scheme of Figure 6 from Section 5.1, the *Updatable* OKMS scheme of Figure 7 does not use an OPRF scheme, e.g. scheme dh-op of Fig. 1, as a black-box: In the UOKMS scheme of Figure 7, the underlying function which maps wraps h to keys dek is ${F'}_{k_c}(h) = H(h^{k_c})$, while the dh-op of Fig. 1 defines $F_k(x) = H(x, (H'(x))^k)$. Even though the server-side in the protocol for oblivious evaluation of ${F'}_{k_c}(\cdot)$ in Fig. 7 is exactly the same as in OPRF for $F_k(\cdot)$ in Fig. 1, the client-side in the two protocols is different, and in particular function argument is not hashed in F'. This lack of inner-hash is important, because even though F' is a PRF, just like F, the oblivious evaluation of F' in Fig. 7 is *not* an Oblivious PRF: Note that a client can use a single interaction with the server to compute ${F'}_{k_c}(\cdot)$ on multiple arguments, e.g. it learns ${F'}_{k_c}(h^i)$ for any i simply by computing $H(v^{i/r})$ in the last step. Formally, the security of this protocol could be modeled as an Oblivious *Weak* PRF, i.e. a single interaction with the server allow the client to compute ${F'}_{k_c}(\cdot)$ on only one argument in the polynomial-sized set of *random* arguments $\{g_1, ..., g_N\}$. We do not formally model this Oblivious Weak PRF primitive, but we use its properties in the security proof of this UOKMS scheme. We note that even though the oblivious decryption protocol in this UOKMS implements this weaker form of OPRF, all the technical benefits discussed in Sections 2 and 4 for dh-op, i.e. extension to *partial*, *threshold*, and *verifiable* OPRF, hold also for the Weak OPRF, i.e. for the oblivious decryption in the UOKMS scheme of Fig. 7.

Efficient UOKMS Scheme. We present our basic Oblivious KMS with Updatable Key Rotation solution in Fig 7. We stress that the enhancements to the base case from Fig. 6, namely, the storage-efficient and threshold schemes, equally apply to the current updatable setting. The computation of the value Δ and its transfer to client C in the case of the threshold schemes use the multi-party technique from Section 6.2. Verification of the server's action are needed

Functions: Function $F : \mathbb{Z}_q \times G \to G$ where $F_k(h) = h^k$; symmetric encryption scheme $\mathsf{Enc}, \mathsf{Dec}$ with keys over $\{0,1\}^\ell$; hash function $H : G \to \{0,1\}^\ell$.

Client keys: Server S stores a client-specific key k_c (for function F) for each client; Client C stores certified value $y_c = g^{k_c}$ where g is a fixed generator of G.

Encryption of object Obj**:** To encrypt Obj under key y_c, pick $s \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, set $h = g^s$ and $\mathsf{dek} = H(y_c{}^s)$, and output ciphertext triple $c = (\mathsf{ObjId}, h, \mathsf{Enc}_{\mathsf{dek}}(\mathsf{Obj}))$.

Decryption of ciphertext $c = (\mathsf{ObjId}, h, e)$**:** C sends $u = h^r$ to S for $r \leftarrow_{\mathrm{R}} \mathbb{Z}_q$; S returns $v = u^{k_c}$ to C only if $u \in G$; C outputs $\mathsf{Obj} = \mathsf{Dec}_{\mathsf{dek}}(e)$ for $\mathsf{dek} = H(v^{1/r})$.

Key rotation and update: To change client's key from k_c to $k_c{}'$, S sends $\Delta = k_c/k_c{}'$ to C. C replaces y_c with $y_c' = (y_c)^{1/\Delta}$ and replaces each ciphertext $c = (\mathsf{ObjId}, h, e)$ in C's storage with $c' = (\mathsf{ObjId}, h' = h^\Delta, e)$.

Fig. 7. Updatable Oblivious KMS Scheme (base case)

only for the decryption operation and these can use the techniques from Section 4[7] A crucial property of this updatable KMS is that the KMS server can delete the key k_c and start using $k_c{}'$ as soon as it has sent Δ to the client. Indeed, the client can transform any wrap that worked with k_c to work with $k_c{}'$ by simply raising the wrap value to the power of Δ.

To show the correctness of the mechanisms in Fig. 7, note that at encryption time, C sets $h = g^s$ for random s, then derives the encryption key dek from y_c^s, and finally stores h. For decryption, C computes h^{k_c} obliviously in interaction with the server and derives dek from this value. Thus, one needs to show that $y_c^s = h^{k_c}$. This is indeed the case since $y_c^s = (g^{k_c})^s = (g^s)^{k_c} = h^{k_c}$.

Regarding the update operation, note that if we denote by h_t and k_t the values of h and k_c, respectively, after t updates (here h_0 denotes the original value of h computed at the time of deriving dek, and k_0 denotes the client's key k_c as it existed at that time), then we can prove inductively that if $h_t^{k_t} = h_0^{k_0}$ (the latter is the value from which dek is derived), then this is also true for $t+1$, namely, $(h_{t+1})^{k_{t+1}} = (h_0)^{k_0}$. Indeed, we have that $h_{t+1} = h_t^{\Delta_{t+1}} = h_t^{k_t/k_{t+1}}$, thus $(h_{t+1})^{k_{t+1}} = (h_t^{k_t/k_{t+1}})^{k_{t+1}} = h_t^{k_t} = h_0^{k_0}$.

6.1 Formal Model for Updatable Oblivious KMS

UOKMS Syntax. Formally, an Updateable Oblivious KMS (UOKMS) scheme is a tuple of algorithms KGen, Enc, UGen, UKey, UEnc and an interactive protocol Dec, where (1) the key generation algorithm KGen on input a security parameter ℓ generates a private public key pair $(\mathsf{sk}, \mathsf{pk})$, (2) the encryption algorithm Enc

[7] The use of dek to decrypt/authenticate data can also serve as validation of the retrieved dek; only if this operation fails one would use the mechanisms from Section 4 (this is especially useful for identifying cheating servers in the threshold case).

on input key pk and plaintext m generates ciphertext c (we write $c \in \mathsf{Enc}(\mathsf{pk}, m)$ if c is a possible output of $\mathsf{Enc}(\mathsf{pk}, m)$), (3) the decryption is implemented via an interactive protocol $\mathsf{Dec} = (\mathsf{Dec.S}, \mathsf{Dec.C})$ between $\mathsf{Dec.S}(\mathsf{sk})$ run by the server and $\mathsf{Dec.C}(\mathsf{pk}, c)$ run by the client, where $\mathsf{Dec.S}$ has no output and $\mathsf{Dec.C}$ outputs m if $c \in \mathsf{Enc}(pk, m)$, (4) the update generation algorithm UGen on input sk generates a new key pair $(\mathsf{sk}', \mathsf{pk}')$ and an update token Δ, (5) the key update algorithm $\mathsf{UKey}(\mathsf{pk}, \Delta)$ outputs the updated public key pk' which matches the public key pk' generated by UGen (if both are executed on the matching $(\mathsf{sk}, \mathsf{pk})$ pair), and (6) the cipheretxt update algorithm $\mathsf{UEnc}(c, \Delta)$ outputs an updated ciphertext c' s.t. $c' \in \mathsf{Enc}(pk', m)$ if $c \in \mathsf{Enc}(pk, m)$.[8] For a set of ciphertexts $C = \{c_i\}_{i=1,\ldots,n}$ we use $\mathsf{UEnc}(C, \Delta)$ to denote the set of corresponding updated ciphertexts, $\{\mathsf{UEnc}(c_i, \Delta)\}_{i=1,\ldots,n}$.

Defining UOKMS Security. We say that UOKMS scheme is *oblivious* if for all efficient $\mathcal{A}$, interaction with $\mathsf{Dec.C}(\mathsf{pk}, c_0)$ is indistinguishable from interaction with $\mathsf{Dec.C}(\mathsf{pk}, c_1)$ for any (pk, c_0, c_1) output by $\mathcal{A}$ s.t. $|c_0| = |c_1|$. As for UOKMS security, we model it with experiments involving adversary $\mathcal{A}$ and simulator SIM shown in Figure 8. We say that UOKMS scheme is *secure* if for all efficient $\mathcal{A}$ there exist an efficient SIM s.t. there is a negligible (in ℓ) difference between the probability that $\mathbf{Exp}^{\mathsf{real}}(\mathcal{A}, \ell)$ outputs 1 and the probability that $\mathbf{Exp}^{\mathsf{ideal}}(\mathcal{A}, \mathsf{SIM}, \ell)$ outputs 1.

Experiment $\mathbf{Exp}^{\mathsf{real}}(\mathcal{A}, \ell)$ in Figure 8 models adversary $\mathcal{A}$'s interaction with the real UOKMS scheme, while $\mathbf{Exp}^{\mathsf{ideal}}(\mathcal{A}, \mathsf{SIM}, \ell)$ models $\mathcal{A}$'s interaction with the simulator SIM which accesses an "ideal" UOKMS scheme. In both games flag corr designates whether $\mathcal{A}$ in a current key epoch corrupts the server (srv) or the client (clt). After the initialization which generates the initial KMS key pair $(\mathsf{sk}, \mathsf{pk})$ and gives pk to $\mathcal{A}$, we assume that $\mathcal{A}$ corrupts either the server or the client in each epoch, modeled by resp. CorClt and CorSrv oracle calls. (We assume w.l.o.g. that $\mathcal{A}$ corrupts one of the parties in each epoch, and that initially $\mathcal{A}$ corrupts the client, which leaks no information beyond public key pk.) Each corruption decision triggers a key update, i.e. the KMS key pair is updated by (re)assigning $(\mathsf{sk}, \mathsf{pk}, \Delta) \leftarrow \mathsf{UGen}(\mathsf{sk})$. We also assume that before each update the client maintains a list of ciphertexts, denoted C, which is updated as $C \leftarrow \mathsf{UEnc}(C, \Delta)$. Adversary $\mathcal{A}$ receives the information corresponding to which party it corrupts: The adversary always gets the new public key pk, but if $\mathsf{corr} = \mathsf{srv}$ then $\mathcal{A}$ also gets the server-held secret key sk (but not the client-held list C), and if $\mathsf{corr} = \mathsf{clt}$ then $\mathcal{A}$ also gets the client-held ciphertext list C (but not the server-held key sk). Moreover, if $\mathcal{A}$ occupies the same party before and after the

[8] UOKMS security does not enforce that the distribution of keys $(\mathsf{sk}', \mathsf{pk}')$ created by UGen is identical to the distribution of keys $(\mathsf{sk}, \mathsf{pk})$ created by KGen, but that is the case in the scheme of Figure 7. Likewise the definition does not impose that ciphertexts updated via $\mathsf{UEnc}(\cdot, \Delta)$ are distributed identically to the fresh ciphertexts generated via $\mathsf{Enc}(\mathsf{pk}, \cdot)$, but that is also the case in the scheme of Figure 7.

update then $\mathcal{A}$ gets the update token Δ, but crucially, $\mathcal{A}$ misses this update token Δ whenever it moves from the server to the client or vice versa.[9]

We assume *active* client corruptions, and in each epoch where $\mathsf{corr} = \mathsf{clt}$ we give $\mathcal{A}$ unfettered access to the ciphertext update algorithm $\mathsf{UEnc}(\cdot, \Delta)$. (Note that if $\mathcal{A}$ corrupts the client both before and after the update then $\mathcal{A}$ holds Δ and can implement this oracle locally, but it is not so when $\mathcal{A}$ corrupts a client after corrupting the server.) In addition, if $\mathsf{corr} = \mathsf{clt}$ then $\mathcal{A}$ can (re)define the client-held list C of ciphertexts which will be updated in the next update. This plays no role if $\mathcal{A}$ continues to corrupt the client during subsequent epochs, because in this case $\mathcal{A}$ could use either Δ or the $\mathsf{UEnc}(\cdot, \Delta)$ oracle to update any ciphertext itself, but if $\mathcal{A}$ corrupts the server in the next epoch then the list C will be updated as $C \leftarrow \mathsf{UEnc}(C, \Delta)$ (and it will keep being updated at each epoch increment), and the adversary will see a modified list C in any epoch in the future when it decides to corrupt the client again. We note that, by contrast, our model for server corruptions is *passive*, and in particular we do not allow $\mathcal{A}$ to change the server's key, to interfere in the update generation or the dissemination of the resulting update token Δ, or to interfere in the interactions of other clients with the update oracle $\mathsf{UEnc}(\cdot, \Delta)$.

Our UOKMS scheme is a public key encryption, so $\mathcal{A}$ can encrypt messages at will, but we include oracle $\mathsf{Enc}(\mathsf{pk}, \cdot)$ in Figure 8 to model the *challenge ciphertexts* in the security experiment. Namely, in the real game, accessing such oracle on input m generates a ciphertext $c \leftarrow \mathsf{Enc}(\mathsf{pk}, m)$, but in the ideal game such ciphertext must be produced by the simulator algorithm SIM on input only $(\mathsf{pk}, |m|)$, while plaintext m is added to the list L of encrypted challenge plaintexts. Adversary $\mathcal{A}$ can also decrypt challenge ciphertexts (or indeed any other ciphertexts) using the decryption oracle $\mathsf{Dec}(\mathsf{sk}, \cdot)$. Because we want to support *oblivious* decryption algorithms, therefore without loss of generality we count each decryption oracle access as an attempt to decrypt some challenge ciphertext. Consequently, in the ideal game we give SIM access to a *single* location on the current list L of challenge plaintexts per each Dec query of $\mathcal{A}$. Observe that we do not create challenge ciphertexts in an epoch where the server is corrupted, because knowledge of the server's private key makes all such ciphertexts insecure. Likewise in the same epoch we disallow $\mathcal{A}$ from accessing an update oracle because then $\mathcal{A}$ could update any challenge ciphertext from a previous epoch to the current one and then decrypt it because $\mathcal{A}$ holds the current private decryption key.

The $\mathbf{Exp}^{\mathsf{real}}$ security game allows any pattern of corruptions *except* corruption of both a client and a server in a single epoch. However, our model of corruptions is *static* in the sense that $\mathcal{A}$ must decide which party to corrupt at the beginning of each epoch. (See also the discussion of other KMS models below.)

[9] Indeed, in the scheme of Figure 7 an old private key sk together with update Δ suffice to derive the *new* private key sk'. Likewise the new key sk' together with Δ suffice to derive the *old* key sk. Either case shows that receiving Δ on transition would imply corrupting both the client and the server in one epoch, in which case no security could be provided.

$\mathbf{Exp}^{\mathsf{real}}(\mathcal{A}, \ell)$

Generate $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}(\ell)$ and initialize C as an empty list and set $\mathsf{corr} \leftarrow \mathsf{clt}$. Experiment output is set to the output of $\mathcal{A}(\mathsf{pk})$ after interaction with the following oracles:

$\mathsf{Enc}(m)$: If $\mathsf{corr} = \mathsf{clt}$ output $\mathsf{Enc}(\mathsf{pk}, m)$;
Dec: If $\mathsf{corr} = \mathsf{clt}$ respond to $\mathcal{A}$ using $\mathsf{Dec.S}(\mathsf{sk})$;
$\mathsf{UEnc}(c)$: If $\mathsf{corr} = \mathsf{clt}$ output $\mathsf{UEnc}(c, \Delta)$;
CorSrv: Set $(\mathsf{sk}, \mathsf{pk}, \Delta) \leftarrow \mathsf{UGen}(\mathsf{sk})$, $C \leftarrow \mathsf{UEnc}(C, \Delta)$;
If $\mathsf{corr} = \mathsf{srv}$ output $(\mathsf{pk}, \mathsf{sk}, \Delta)$ else output $(\mathsf{pk}, \mathsf{sk})$ and set $\mathsf{corr} \leftarrow \mathsf{srv}$;
CorClt: Set $(\mathsf{sk}, \mathsf{pk}, \Delta) \leftarrow \mathsf{UGen}(\mathsf{sk})$, $C \leftarrow \mathsf{UEnc}(C, \Delta)$;
If $\mathsf{corr} = \mathsf{clt}$ output (pk, C, Δ) else output (pk, C) and set $\mathsf{corr} \leftarrow \mathsf{clt}$;
Let $\mathcal{A}$ (re)define set C;

$\mathbf{Exp}^{\mathsf{ideal}}(\mathcal{A}, \mathsf{SIM}, \ell)$

Let $\mathsf{SIM}(\ell)$ generate pk and initialize empty challenge plaintext list L. Experiment output is set to the output of $\mathcal{A}(\mathsf{pk})$ after interaction with the following oracles:

$\mathsf{Enc}(m)$: If $\mathsf{corr} = \mathsf{clt}$ add m to L, output $\mathsf{SIM}(\mathsf{enc}, |m|)$;
Dec: If $\mathsf{corr} = \mathsf{clt}$ respond to $\mathcal{A}$ using $\mathsf{SIM}(\mathsf{dec})$ and let SIM access *one* chosen location in list L;
$\mathsf{UEnc}(c)$: If $\mathsf{corr} = \mathsf{clt}$ output $\mathsf{SIM}(\mathsf{upd}, c)$;
CorSrv: Set $\mathsf{corr} \leftarrow \mathsf{srv}$, output $\mathsf{SIM}(\mathsf{srv})$;
CorClt: Set $\mathsf{corr} \leftarrow \mathsf{clt}$, output $\mathsf{SIM}(\mathsf{clt})$, get C from $\mathcal{A}$;

Fig. 8. Security Games for Updatable Oblivious KMS

Comparison to Other Updatable Encryption Models. Several recent works considered Updatable KMS under the name *Updatable* Encryption or Encryption with *Key Rotation* [8,9,21,40]. Our notion corresponds to the *ciphertext-independent* notion of Everspaugh et al. [21] and Lehmann and Tackmann [40], but there are several differences between our model and [21,40]. Firstly, the security notions of [8,9,21,40] all model only IND-CPA notion of encryption indistinguishability and make *no security claims in the presence of decryption oracle*. Restriction to IND-CPA security might appear natural since encryption updates must rely on some form of ciphertext malleability, which precludes standard notion of IND-CCA security. However, our simulation-based security notion does capture security in the presence of decryption oracle with a simple counting method which enforces that Q accesses to the decryption oracle allow for learning plaintext information in at most Q challenge ciphertexts. The second major difference is that ours is the first treatment of Updatable Encryption with *oblivious* decryption procedure, which indeed necessitates this "counting-

based" notion of security in the presence of decryption oracle. (We note that Green [26] formalized IND-CCA security of encryption with oblivious decryption, but without updates.) On the other hand, [40] consider an *adaptive* model of corruption, including in particular post-execution corruption of keys from any past epoch, while our corruption model is *static*. Indeed, a combination of an adaptive model *and* a decryption oracle seems to present significant technical challenges, akin to adaptive security in proactive cryptosystems, see e.g. [13,2,41]. There are many other differences of seemingly lower importance, e.g. we support *public key* encryption while [21,40] support symmetric-key encryption, and [21] enforces ciphertext integrity, a property which is relevant only to symmetric-key encryption. Also, both [21,40] model update indistinguishability, which we do not consider formally, although we think that our scheme could be easily adapted to support it.

Security of our UOKMS Scheme. The UOKMS scheme shown in Figure 7 is information-theoretic *oblivious*, as is the OPRF protocol dh-op on which the Decryption protocol in Fig. 7 is based, but the *security* of this scheme according to the above real/ideal UOKMS security notion requires the following computational assumption:

One-More DH with Inverse Oracle (OMDH-IO) Assumption. For any PPT $\mathcal{A}$ the following probability is negligible:

$$\mathsf{Prob}[\mathcal{A}^{(\cdot)^k,(\cdot)^{1/k}}(g, g^k, g_1, \ldots, g_N) = \{(g_{j_s}, g_{j_s}^k)\}_{s=1,\ldots,Q+1}]$$

with the probability going over random k in $\mathbb{Z}_q$, random choice of group elements $g_1, \ldots, g_N$ in $G = \langle g \rangle$, and $\mathcal{A}$'s randomness, and where $(\cdot)^k$ and $(\cdot)^{1/k}$ are exponentiation oracles, and Q is the number of queries that $\mathcal{A}$ poses to the $(\cdot)^k$ oracle.

In addition to the OMDH-IO assumption and the Random Oracle Model (ROM) for hash function H, the security proof also relies on a technical property of symmetric encryption $(\mathsf{Enc}, \mathsf{Dec})$ which we call *adaptive security*, defined as follows:

Adaptive Security of SKE. We call symmetric encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ *adaptively secure* if for any PPT $\mathcal{A}$ there exists PPT SIM s.t. $\mathcal{A}$'s interaction in the following real and ideal games is indistinguishable: (1) In the real game $\mathcal{A}$ interacts with oracle Enc and Reveal, where oracle Enc on input (i, m) picks random key k_i and outputs $c = \mathsf{Enc}(k_i, m)$ while oracle Reveal on input i reveals k_i; (2) In the ideal game $\mathcal{A}$ interacts with the stateful algorithm SIM, s.t. when $\mathcal{A}$ sends (i, m) as its Enc oracle query, SIM computes c given $(i, |m|)$ as input, and when $\mathcal{A}$ sends i as its Reveal oracle query, SIM produces k_i given (i, m) as input.

Theorem 2. *The UOKMS scheme in Figure 7 is unconditionally* oblivious *and it is* secure *under the OMDH-IO assumption in ROM if the symmetric encryption scheme* $(\mathsf{Enc}, \mathsf{Dec})$ *is adaptively secure.*

Proof: See paragraph below for an intuition and Appendix C for the full proof.

The following corollary follows because common encryption modes including CTR and CBC satisfy the adaptive security property in the Ideal Cipher model. This is easy to see: For $|m|$ equal to n blocks for block cipher E, SIM chooses random IV and sets ciphertext $e = (IV, e_1, ..., e_n)$ where e_i's are all random blocks, and when SIM gets $m = (m_1, ..., m_n)$ it sets n input/output points of $E(k, \cdot)$: For CTR SIM sets $E(k, IV + i) = m_i \oplus e_i$ for all i, and for CBC it sets $E(k, m_i \oplus e_{i-1}) = e_i$ for all i and $e_0 = IV$. Either way by randomness of e_i's this sets $E(k, \cdot)$ outputs on n points to random values, it creates collisions in $E(k, \cdot)$ with negligible probability, and by randomness of k there is negligible probability that any points of $E(k, \cdot)$ were queried before.

Corollary 1. *The UOKMS scheme in Figure 7 is* secure *under the OMDH-IO assumption in the Ideal Cipher Model and ROM if the symmetric encryption is implemented using CTR or CBC modes.*

Proof rationale. The proof of Theorem 2 is in Appendix C, but here we briefly explain why security reduces to OMDH-IO. If $(k_i, y_i = g^{k_i})$ denotes the KMS key for a given client in epoch i, the reduction will set (k_0, y_0) as the OMDH-IO challenge key (k, g^k), and compute all keys for rounds when $\mathcal{A}$ corrupts the client as $y_i = y^{1/\delta_i}$ where $\delta_i = \prod_{j \le i} \Delta_j$ where Δ_j's are all chosen at random. If the adversary corrupts the server in rounds $j+1, ..., t-1$ then the reduction will pick random server keys $k_{j+1}, ..., k_{t-1}$ and use random "super-update" value $\Delta_{j+1,t}$ in place of $\Delta_{j+1} \cdot ... \cdot \Delta_t$ in the equation for δ_i for $i \ge t$. The same δ_i values suffice to update the ciphertexts written by $\mathcal{A}$ in client-held ciphertext list C, and they also suffice to translate $(\cdot)^k$ OMDH challenges $g_1, ..., g_N$ into epoch-i ciphertexts in $\mathsf{Enc}(\cdot)$ oracle calls, and to translate epoch-i decryption queries $\mathsf{Dec}(\cdot)$ to OMDH queries $(\cdot)^k$. However, when $\mathcal{A}$ corrupts the server in round $t-1$ and then corrupts the client in round t, it gets random key k_{t-1} and then gains access to the update oracle which implements exponentiation $(\cdot)^{\Delta_t}$ for $\Delta_t = k_{t-1}/k_t$. Since k_{t-1} is known, this is equivalent to gaining access to exponentiation oracle $(\cdot)^{1/k_t}$, and since k_t is linked to the OMDH challenge key k as $k_t = k/\delta_i$, this is in turn equivalent to accessing exponentiation oracle $(\cdot)^{1/k}$. Thus the security of the UOKMS scheme in Figure 7 *requires* hardness of One More Diffie-Hellman in the presence of the inversion oracle $(\cdot)^{1/k}$, but the reduction sketched above implies that the OMDH-IO assumption also *suffices* for its security. The full proof is shown in Appendix C.

6.2 Threshold Updates

The Updatable Oblivious KMS of Fig. 7 can be efficiently implemented in the *threshold setting* using the same share conversion technique of [18] shown in the $\mathsf{tdh\text{-}pop}$ protocol in Fig. 4. However, note that in this setting, keys $k_c, k_c{}'$ are *secret-shared* between the servers, so also the update value $\Delta = k_c/k_c{}'$ needs to be computed in a distributed way such that only the client learns Δ and

no one learns (anything about) the keys $k_c, k_c{}'$. For obtaining such distributed computation of Δ we resort to two tools from threshold cryptography. The first is a protocol that given the Shamir sharing of a secret a and a sharing of a secret b, generates a sharing of the product $a \cdot b$ without learning anything about either secret or their product. We denote such protocol by $\mathsf{ProdShare}(a, b)$ and we recall an implementation from [24] in Appendix D. The second tool allows for the joint generation of a secret sharing of a random secret (e.g., [46] or Fig. 7 of [24]). Using these tools we obtain the following procedure.

Client computation of $\Delta = k_c/k_c{}'$ from sharings of k_c and $k_c{}'$. Assuming servers $S_1, \ldots, S_n$ secret-share keys k_c and $k_c{}'$ in (n, t)-threshold Shamir secret-sharing, servers $S_1, \ldots, S_n$ first generate a joint (n, t)-threshold sharing $\rho_1, \ldots, \rho_n$ of a *secret* value ρ which is randomly chosen in $\mathbb{Z}_q$. Then they run the protocol $\mathsf{ProdShare}(k_c, \rho)$ and $\mathsf{ProdShare}(k_c{}', \rho)$, generating (n, t) sharings $(r_1, \ldots, r_n)$ and $(r'_1, \ldots, r'_n)$ of the products $\rho \cdot k_c$ and $\rho \cdot k_c{}'$, respectively. Finally, each server S_i sends to the client the shares r_i, r'_i, and the client reconstructs $r = \rho \cdot k_c$ and $r' = \rho \cdot k_c{}'$ and computes $\Delta = r/r'$.

Security of this protocol follows from (1) security of $\mathsf{ProdShare}$; (2) the fact that if $k_c, k_c{}' \neq 0$ then r, r' are distributed as two random elements in $\mathbb{Z}_q^*$ subject to the constrain that $r/r' = k_c/k_c{}'$; and (3) the fact that the (n, t)-threshold secret-sharings $(r_1, \ldots, r_n)$ and $(r'_1, \ldots, r'_n)$ reveal no additional information except for values r, r' that can be reconstructed (by the client in our application) from this sharing.

In the resulting protocol servers $S_1, \ldots, S_n$ calculate and transmit to the client the value $\Delta = k_c/k_c{}'$ where keys $k_c, k_c{}'$ are *secret-shared* between these servers, and in fact their sharing is non-interactively obtained using the exact same share conversion technique we use to derive the shares of the Threshold (p)OPRF key in $\mathsf{tdh\text{-}pop}$.

7 Implementation and Performance

We report on implementation and performance of the OKMS and UOKMS schemes from Section 5.1 (Fig. 6) and Section 6 (Fig. 7), respectively.

Microbenchmarks. Implementations of all necessary client and server operations were written in C++ using the OpenSSL library (version 1.1.1-pre5) to provide cryptographic functionality. Performance tests were conducted on a machine with an Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz having 15 GB of memory. The implementation was compiled with the gcc compiler with optimization level 3.

The following tables detail the run times of each operation averaged over 10,000 trials. These tests used only a single thread and CPU core; results could be improved by taking advantage of the parallelism that is readily possible with share conversion (expression (3)) or by performing these operations concurrently across multiple CPU cores.

In these tests, all elliptic curve operations were based on NIST P-256, using AES-256 keys for CMAC operations. Field operations (for Shamir, and blinding factors) were defined over the prime order of NIST P-256. Hashing to the curve was performed using SHA-256 together with the constant time *Simplified SWU* algorithm[11].

OKMS Client Operations (Single Thread)		
	Wrap	Unwrap
Hash to Curve	58.26	58.26
Generate Blind	1.58	1.58
Apply Blind	68.07	68.07
Create Challenge	83.27	-
Interpolate Result	7.67	7.67
Inverse Blind	16.95	16.95
Remove Blind	68.07	68.07
Verify Response	106.67	-
Total Time (μs)	410.53	220.60
Operations / Second	2,435.86	4,533.14

Fig. 9. Client operation time and Op/s in OKMS

Client operations in the OKMS scheme are shown in Fig. 9 for both encryption ("wrapping") and decryption ("unwrapping"). The two operations differ only in that interactive verifiability is performed for wrap operations, while for unwrapping (the more common operation) regular symmetric key verification suffices.

In comparison to the client microbenchmark of pOPRF operations presented in [20], which required 5500 μs for the client to perform a verifiable pOPRF, our result of 410.53 μs is 13.40 times faster. This result was achieved on equivalent hardware as reported in the Pythia paper: a third-generation Intel Xeon with 8 cores at 2.9 GHz.

Server operations, measured for the storage-efficient threshold variant of this (U)OKMS scheme (implemented with protocol tdh-pop) are shown in Fig. 10 for different pairs $(n, t+1)$. This includes performing share conversion, computing a Lagrange coefficient, multiplying the server's k_i by the coefficient, and performing an exponentiation (EC scalar multiply) in the curve for each input provided by the client (the wrap operation is more costly as it includes an additional exponentiation to support the interactive verification procedure from Section 4). The percent of time the server spends performing share conversion in a wrap varies from 3.44% on the low end (3-of-5) to 85.63% on the high end for the 5-of-15 sharing. Caching share conversion results would improve performance radically in the high-end cases.

In comparison to the server microbenchmark of pOPRF presented by [20], which required 4000 μs to perform a verifiable pOPRF and 1500 μs to perform an

unverified pOPRF, our equivalent operations of *wrap* and *unwrap* require 136.93 μs and 68.87 μs respectively, yielding a 29x and 21x performance improvement (again on equivalent hardware).

(U)OKMS Server Operations (Single Thread)		
(t+1)-of-N	Wraps / Second	Unwraps / Second
1-of-1	7,302.47	14,519.13
3-of-5	6,946.99	13,178.35
4-of-7	6,427.64	11,426.89
6-of-11	2,736.95	3,363.56
7-of-13	1,107.56	1,197.86
5-of-15	1,033.98	1,112.26
4-of-20	1,054.51	1,136.05
3-of-40	1,266.33	1,385.77

Fig. 10. (U)OKMS Server Op/s for practical threshold parameters

Client performance in the UOKMS scheme (Sec. 6) is shown in Fig. 11. This setting benefits from being able to perform wrap operations without server involvement, and can further benefit from precomputation tables for exponentiation of g and g^k. In our testing, precomputation provided more than a 600% speed up (11.33 vs. 68.20 μs). We summarize the number of operations per second the client can perform in the UOKMS.

UOKMS Operations (Single Thread)		
	Time (μs)	Operations / Second
Wrap	24.24	41,261.68
Unwrap	162.33	6,160.14
Update	68.07	14,691.77

Fig. 11. Client operation time and Op/s in UOKMS

(U)OKMS Server. To evaluate performance and scalability we hosted our (U)OKMS server implementation on Amazon's Elastic Compute Cloud (EC2)[4] using a *c4.2xlarge* instance type. This instance type provides 8 virtual CPUS with an Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz having 15 GB of memory and was the same instance type used to obtain the microbenchmark numbers above.

Requests to this server were issued over HTTP and the web server, *nginx*, was configured with 8 worker processes (one per CPU). OKMS functionality was added to this web server as a natively compiled module which used the OpenSSL library (version 1.1.1-pre5) to provide cryptographic functionality. The server ran Ubuntu 16.04 as its operating system.

Throughput. To measure throughput a client machine (also *c4.2xlarge*) was deployed in the same Amazon Web Service (AWS) availability zone as the server. We used the HTTP load generating tool *hey* to measure the throughput for each scheme. hey was configured with a concurrency level of 80 and all results were averaged over 50,000 requests. All requests were for an unwrap operation and were sent over HTTPS using (TLS 1.2 with ECDHE-ECDSA-AES256-GCM-SHA384). The server used a self-signed certificate with an EC key on the NIST P-256 curve. Computation time dominates in the LAN setting due to almost negligible network latency, with the CPU cores reaching near 100% utilization during the LAN throughput tests. To gauge the limits of the server performance, client-side operations of blinding and verification were not performed by the load generator.

For each scheme, we tested with session KeepAlive on and off. When off, a new TCP connection and TLS session must be negotiated for each request. When on, the connection setup costs are amortized over all requests, which is in line with a client that must unwrap many keys.

The table in Figure 12 details the observed throughput in requests per second (RPS) for the various schemes and two KeepAlive configurations (all over TLS). We compare these schemes to a static page as a baseline.

(U)OKMS Server Throughput (8 CPU cores)		
Scheme	KeepAlive	No KeepAlive
Static Page	65,018.02	6,462.29
OPRF / Cached pOPRF	32,094.31	6,349.80
3-of-5 pOPRF	31,053.97	6,833.13
6-of-11 pOPRF	12,511.51	5,122.76

Fig. 12. (U)OKMS Server Requests/s on EC2 instance (LAN setting)

We observe that for the *No KeepAlive* configuration, the cost of creating the new connection and establishing the TLS session dominates resulting in very little difference in RPS between the schemes. For the *KeepAlive* configuration, throughput is significantly better, achieving over 30,000 RPS for the cached pOPRF case and the 3-of-5 share conversion case. The 6-of-11 is the slowest as it must perform a more expensive share conversion involving 252 PRF operations, but still achieves over 10,000 RPS. Thus our (U)OKMS implementation can handle a large number of clients with a single server. For comparison, Google processes 40,000 search queries per second[31]. If needed, the implementation can be scaled with standard techniques, such as deploying a greater number of servers.

The RPS achieved by the server with its 8 CPU cores does not reflect the idealized rate of 8x the Unwraps/Second given in Fig. 10 for the 3-of-5 and 6-of-11 configurations. The server achieves improvement ratios of 2.4x to 3.7x over the microbenchmarks respectively. Falling short of the ideal is expected given a

constant overhead introduced by HTTPS and the web server. Different improvement ratios are also explained by this overhead. The more time-consuming the case is (e.g., 6-of-11), the less significant the constant overhead is and the better the improvement ratio will be.

Latency. We measured wide-area network latency for various configurations and access patterns. In our test the server was located in the Amazon's *us-east-1* availability zone which is in Northern Virginia while the client machine was located in Chicago and had an Intel(R) Core(TM) i7-4980HQ CPU @ 2.80GHz having 16 GB of memory.

For a base-line, round trip time (RTT) was measured using the *ping* utility. All measurements below are the average of 10 operations. For the 10-request cases, keep-alive was used. Without keep-alive, the 10-request cases would be equal to 10 times the respective single request case.

(U)OKMS Server Latency		
Operation	Latency (ms)	Latency in RTTs
ping (baseline)	25.93	1.00
1 HTTP request	84.60	3.26
1 HTTPS request	129.30	4.99
10 HTTP requests	310.40	11.97
10 HTTPS requests	378.10	14.58

Fig. 13. Observed Latencies over WAN

In Figure 13 we see the total time for the (U)OKMS server to process 6-of-11 pOPRF requests for various access patterns. In this view, total perceived time per operation is dominated by network latencies. Even without the TLS handshake, a single HTTP requests requires 3.26 RTTs (84.6 milliseconds) to complete, while introducing TLS for the HTTPS case increases this time to nearly 5 RTTs. Reusing an established HTTP or HTTPS connection greatly reduces the per-request time, with the 10-request cases being approximately equal to the single request case plus 9 RTTs.

Since client and server CPU processing times are a sub-millisecond per request, WAN latencies are expected to dominate in any cross-regional network.

Conventional engineering improvements such as parallelization and concurrency can reduce latency for the multiple request access pattern. Further latency improvements can be obtained with less expensive connection-establishment protocols, e.g., TLS with session resumption, TLS 1.3, QUIC [38], or a custom protocol. We note that when using the pOPRF verifiability mechanism in our KMS protocols, TLS or other channel security mechanisms are unnecessary for the confidentiality or integrity of returned keys. In principle a secure protocol could be designed that required only a single round trip between the client and server, yielding a total access latency for unwrapping one or more keys approximately to a single RTT.

8 Conclusions

We have presented very efficient realizations of partially-oblivious PRFs (pOPRF) with over an order of magnitude performance improvements over the previous Pythia pOPRF construction from [20] and have showed how to extend these schemes to threshold implementations. We demonstrate the utility of the schemes by building a novel key management system (KMS) on top of our pOPRFs. The resultant Oblivious KMS (OKMS) offers major improvements in security relative to traditional systems. Whereas in standard wrapping-based KMS across the industry [3,42,30,25] the server learns all of its clients' encryption keys, in our OKMS, keys and data object identifiers are hidden from the KMS with unconditional security (in the DH-based schemes). Additionally, the solution offers unconditional security for key transport, enables forward security, provides key verifiability, and reduces storage. Our distributed (threshold) implementation of such a service additionally protects against the corruption of a subset of servers.

We further extend our (threshold) OKMS into an Updatable OKMS to support updatable encryption so that upon the periodic change of a client's pOPRF key by the server, a short update token Δ sent from server to client allows the latter to update its state so that all its encrypted data can be decrypted through the updated pOPRF key but none of it can be decrypted using past pOPRF keys (hence providing forward secrecy). The client update operation requires no interaction with the server (except for receiving the update token), and involves no encryption/decryption of data by the client. Our techniques improve in both efficiency and security on several recent works on updatable encryption, including the work of [21] from Crypto'17 and [40] from Eurocrypt'18. Improvements include the use of a single exponentiation per updated item vs. two exponentiations per item in [21,40]. Morevoer, our KMS scheme is *oblivious*, i.e. the server does not learn the decrypted messages (or even which ciphertexts are decrypted). Finally, we show much stronger security of encryption in the form of CCA-type security against adversaries who can decrypt (and update) adaptively-chosen ciphertexts. By contrast, [21,40] show only CPA-type security for KMS schemes with efficient updates. In particular, in the context of Updatable KMS (whose goal is recovery from adaptive corruptions), we guarantee security even if an attacker observes decryption of ciphertexts.

We prove our protocols in the strongest security model: universal composability (UC). In particular, while the previous security definition of Partial OPRF (pOPRF) from [20]) was somewhat ad-hoc, here we define this primitive via a UC functionality that inherits the strength of the UC OPRF definition from [35,36]. We additionally provide an analysis of the security properties of the UOKMS system as listed above, improving on definitions (and security) from [8,9,21,40] to capture a CCA-like notion of updatable encryption and oblivious operations.

Finally, we reported on our implementation work and performance testing showing the practicality of our solutions and the major performance improvement relative to prior pOPRF constructions. We show gains of 13x for client performance and 29x for the server relative to Pythia [20]. This efficiency trans-

lates into our implementation of the OKMS and UOKMS services, including their threshold versions. In all, we believe that the security properties of our schemes together with their excellent performance make them great candidates for real-world deployment as a replacement to existing storage and key management systems.

References

1. R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 86–97, New York, NY, USA, 2003. ACM.
2. J. F. Almansa, I. Damgård, and J. B. Nielsen. Simplified threshold rsa with adaptive and proactive security. In S. Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, pages 593–611, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
3. Amazon Web Services. Aws key management service cryptographic details, 2016. `https://d1.awsstatic.com/whitepapers/KMS-Cryptographic-Details.pdf`.
4. Amazon Web Services. Aws elastic compute cloud, 2018. `https://aws.amazon.com/ec2/`.
5. M. Bellare, J. A. Garay, and T. Rabin. Fast batch verification for modular exponentiation and digital signatures. In K. Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 236–250. Springer, Heidelberg, May / June 1998.
6. M. Bellare, S. Keelveedhi, and T. Ristenpart. Message-locked encryption and secure deduplication. In T. Johansson and P. Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 296–312. Springer, Heidelberg, May 2013.
7. A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In Y. Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 31–46. Springer, Heidelberg, Jan. 2003.
8. D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan. Key homomorphic prfs and their applications. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 410–428, 2013.
9. D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan. Key homomorphic prfs and their applications. *IACR Cryptology ePrint Archive*, 2015:220, 2015.
10. D. Boneh, B. Lynn, and H. Shacham. Short signatures from the Weil pairing. In C. Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 514–532. Springer, Heidelberg, Dec. 2001.
11. E. Brier, J.-S. Coron, T. Icart, D. Madore, H. Randriam, and M. Tibouchi. Efficient indifferentiable hashing into ordinary elliptic curves. Cryptology ePrint Archive, Report 2009/340, 2009. `http://eprint.iacr.org/2009/340`.
12. J. Camenisch and A. Lehmann. Privacy for distributed databases via (un)linkable pseudonyms. Cryptology ePrint Archive, Report 2017/022, 2017. `http://eprint.iacr.org/2017/022`.
13. R. Canetti, R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Adaptive security for threshold cryptosystems. In M. Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 98–116, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

14. D. Cash, S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for Boolean queries. In R. Canetti and J. A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 353–373. Springer, Heidelberg, Aug. 2013.
15. D. Chaum. Zero-knowledge undeniable signatures. In I. Damgård, editor, *EUROCRYPT'90*, volume 473 of *LNCS*, pages 458–464. Springer, Heidelberg, May 1991.
16. D. Chaum and T. P. Pedersen. Wallet databases with observers. In E. F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 89–105. Springer, Heidelberg, Aug. 1993.
17. S. S. M. Chow, C. Ma, and J. Weng. Zero-knowledge argument for simultaneous discrete logarithms. In M. T. Thai and S. Sahni, editors, *Computing and Combinatorics*, pages 520–529, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
18. R. Cramer, I. Damgård, and Y. Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In J. Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 342–362. Springer, Heidelberg, Feb. 2005.
19. E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *International Conference on Financial Cryptography and Data Security*, pages 143–159. Springer, 2010.
20. A. Everspaugh, R. Chaterjee, S. Scott, A. Juels, and T. Ristenpart. The pythia PRF service. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 547–562, Washington, D.C., 2015. USENIX Association.
21. A. Everspaugh, K. G. Paterson, T. Ristenpart, and S. Scott. Key rotation for authenticated encryption. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, pages 98–129, 2017.
22. W. Ford and B. S. Kaliski Jr. Server-assisted generation of a strong secret from a password. In *9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2000)*, pages 176–180, Gaithersburg, MD, USA, June 4–16, 2000. IEEE Computer Society.
23. M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In J. Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, Feb. 2005.
24. R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and fact-track multiparty computations with applications to threshold cryptography. In B. A. Coan and Y. Afek, editors, *17th ACM PODC*, pages 101–111. ACM, June / July 1998.
25. Google Cloud. Google cloud key management service, 2018. `https://cloud.google.com/kms/`.
26. M. Green. Secure blind decryption. In *Proceedings of the 14th International Conference on Practice and Theory in Public Key Cryptography Conference on Public Key Cryptography*, PKC'11, pages 265–282, Berlin, Heidelberg, 2011. Springer-Verlag.
27. C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In R. Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 155–175. Springer, Heidelberg, Mar. 2008.
28. A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In D. Coppersmith, editor, *CRYPTO'95*, volume 963 of *LNCS*, pages 339–352. Springer, Heidelberg, Aug. 1995.
29. B. A. Huberman, M. K. Franklin, and T. Hogg. Enhancing privacy and trust in electronic communities. In *EC*, 1999.
30. IBM. Ibm key protect, 2018. `https://console.bluemix.net/catalog/services/key-protect`.

31. Internet Live Stats. Google Search Statistics, 2018. http://www.internetlivestats.com/google-search-statistics/.
32. M. Ito, A. Saito, and T. Nishizeki. Secret sharing schemes realizing general access structure. In *Proc. IEEE Global Telecommunication Conf. (Globecom'87)*, pages 99–102, 1987.
33. S. Jarecki, C. S. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In A.-R. Sadeghi, V. D. Gligor, and M. Yung, editors, *ACM CCS 13*, pages 875–888. ACM Press, Nov. 2013.
34. S. Jarecki, A. Kiayias, and H. Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In P. Sarkar and T. Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253. Springer, Heidelberg, Dec. 2014.
35. S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu. Highly-efficient and composable password-protected secret sharing (or: how to protect your bitcoin wallet online). In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 276–291. IEEE, 2016.
36. S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. In D. Gollmann, A. Miyaji, and H. Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 39–58. Springer, Heidelberg, July 2017.
37. S. Jarecki and X. Liu. Fast secure computation of set intersection. In J. A. Garay and R. D. Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 418–435. Springer, Heidelberg, Sept. 2010.
38. Jim Roskind. QUIC: Multiplexed stream transport over UDP, 2013. Google working design document.
39. R. W. F. Lai, C. Egger, D. Schröder, and S. S. M. Chow. Phoenix: Rebirth of a cryptographic password-hardening service. In *USENIX Security Symposium*, 2017.
40. A. Lehmann and B. Tackmann. Updatable encryption with post-compromise security. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, pages 685–716, 2018.
41. A. Y. Lindell. Adaptively secure two-party computation with erasures. In M. Fischlin, editor, *Topics in Cryptology – CT-RSA 2009*, pages 117–132, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
42. Microsoft Azure. Azure key vault, 2018. https://docs.microsoft.com/en-us/azure/key-vault/key-vault-overview.
43. M. Naor, B. Pinkas, and O. Reingold. Distributed pseudo-random functions and KDCs. In J. Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 327–346. Springer, Heidelberg, May 1999.
44. M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *38th FOCS*, pages 458–467. IEEE Computer Society Press, Oct. 1997.
45. A. Patel and M. Yung. Fully dynamic password protected secret sharing, 2017. manuscript.
46. T. P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract) (rump session). In D. W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 522–526. Springer, Heidelberg, Apr. 1991.
47. K. Sakurai and Y. Yamane. Blind decoding, blind undeniable signatures, and their applications to privacy protection. In *Proceedings of the First International Workshop on Information Hiding*, pages 257–264, London, UK, UK, 1996. Springer-Verlag.

48. M. Shirvanian, S. Jarecki, H. Krawczyk, and N. Saxena. Sphinx: A password store that perfectly hides passwords from itself. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 1094–1104. IEEE, 2017.

A Non-interactive Verification for (Partially) Oblivious PRFs

Here we complement the material in Section 4 by recalling two standard techniques for exponentiation verification that apply to the verifiability of our pOPRF constructions, including the threshold variants.

Verification via Non-Interactive Zero-Knowledge (NIZK). The standard approach to verification uses non-interactive zero-knowledge proofs which for our application means proofs of equality of discrete logarithms. For example, in the case of protocol dh-op in Fig. 1, the server would produce a NIZK proof showing that its response b (specified as a^k) satisfies $\text{dlog}_a(b) = \text{dlog}_g(v)$ where v is the OPRF verification key.

The most efficient NIZK protocol for this task is due to Chow et al [17] which for the single-server case only takes one 2-element multi-exp for the server (it costs 1.17 regular exponentiation) and one 4-element multi-exp for the client (at the cost about 1.25 regular exponentiation), making it somewhat more efficient than the above interactive solution for the client. However, for the threshold case, it requires individual zero-knowledge proofs from the servers and their verification. Performance can be optimized via batch verification [5] which reduces the verification of the individual proofs to two multi-exponentiation with many factors. A very optimized implementation may be competitive with the cost of the interactive proof but at higher implementation complexity.

Verification via Signature-based OPRFs. At the end of Section 2, we mentioned BLS signatures [10] whose signature operation, an exponentiation over a pairings-friendly group G_1, can serve as an alternative implementation of our DH-based OPRFs and pOPRF schemes. While its use of bilinear groups results in performance degradation relative to the most efficient elliptic curve groups, the advantage of this scheme is that verification of exponentiation can be done via the signature verification procedure of the BLS scheme. The scheme requires no additional action from the server but verification is costly as it involves two pairing operations for the client. This scheme works in the single-server case by verifying the response $b = a^k$ from the server and also in the threshold case where individual exponentiations $b_i = a_i^k$ can be verified against a certified value $v_i = g^{k_i}$. More precisely, the client would first interpolate the received values b_i to obtain $b = a^k$ and then use BLS verification against the aggregated key g^k. Only if this fails would the client verify the individual pieces b_i. We note that other blind signature schemes, e.g., RSA [19,7], can be adapted for a similar use as above.

B Threshold Password Onion Scheme

The Pythia system [20] utilizes a variant of the pOPRF notion[10] to implement a "password hardening" system. Their application (very different from the one in the previous subsection) considers the typical practice of password servers that store a salt-hash pair for each user where the hash value is computed on a password-salt pair using a deterministic hash function. The compromise of a salt-hash pair can then be used to mount an offline dictionary attack on the password (an attack responsible for the compromise of billions of passwords). Pythia avoids this vulnerability by replacing the deterministic hash with a pOPRF computation carried by a pOPRF service (to which we will refer to as Pythia) that is separate from the server storing the salt-hash values. When the latter server needs to compute a hash value for verifying a password, it queries Pythia on oblivious inputs (password, salt). An attacker learning a salt-hash pair cannot mount an attack on the password value without talking directly to Pythia.

This application, referred in [20] as "password onion", requires setting fine-granular rate limitations for which the Pythia system uses non-oblivious inputs. In addition, in order to serve multiple clients, Pythia derives the per-client pOPRF keys from a single master key via a regular PRF. This implements a SpOPRF as in expression (1), except that the Pythia function allows for a non-oblivious input *in addition* to the value y, namely, their scheme has the form $F'_K(x, z, y) = F_{f_K(y)}(x, z)$ where both y and z are non-oblivious. Clearly, the password onion application would enjoy additional security if the Pythia server could be distributed through a threshold implementation. Our tdh-pop scheme from Fig. 4 can be used in this case too, except that one would replace the underlying dh-op scheme (Fig. 1) with the Pythia function defined as $e(H_1(z), H_2(x))^k$ where e is a bilinear map $e : G_1 \times G_2 \to G_T$, and H_1, H_2 are hash functions. Note that while this scheme is significantly more computationally expensive than dh-op due to the use of pairings and exponentiation in G_T, its cost may be justified in applications that require an extra non-oblivious input.

Recently, Lai et al. [39] have shown a more efficient single-server solution than [20] for the specific password hardening application that motivated the Pythia system in the first place. However, we don't know of any solution (including the one from [39]) that works in the multi-server case (with per-client key derivation) other than the above adaptation of tdh-pop to the Pythia function.

C Security Proof for Updatable Oblivious KMS

Below we include the proof of Theorem 2 from Section 6.1 about security of the Updatable Oblivious KMS scheme shown in Figure 7 in Section 6.

proof Below we show an efficient simulator algorithm SIM which having access to (any) adversary algorithm $\mathcal{A}$, interacts with the ideal UOKMS game $\mathbf{Exp}^{\mathsf{ideal}}$.

[10] The Pythia application requires key updates similar to those used in the updatable encryption scheme of Sec. 6, hence they cannot use the outer hashing of the DH-based schemes from Sec. 2, resulting in a weaker form of OPRF.

We will then re-write SIM as a reduction algorithm $\mathcal{R}$ s.t. if $\mathcal{A}$ has ϵ advantage in distinguishing an interaction with the real UOKMS game $\mathbf{Exp}^{\mathsf{real}}$ and an interaction with SIM and $\mathbf{Exp}^{\mathsf{ideal}}$, i.e. if

$$\epsilon = |\ \Pr[1 \leftarrow \mathbf{Exp}^{\mathsf{real}}(\mathcal{A}, \ell)] - \Pr[1 \leftarrow \mathbf{Exp}^{\mathsf{ideal}}(\mathcal{A}, \mathsf{SIM}, \ell)]\ \ |$$

then reduction $\mathcal{R}$, given access to $\mathcal{A}$, has the same probability ϵ of solving the OMDH-IO problem. It follows that under the OMDH-IO assumption quantity ϵ must be negligible, which implies that the UOKMS scheme is secure.

The proof relies on the ROM model for function $H : G \rightarrow \{0,1\}^{\ell}$ used in UOKMS scheme in Figure 7. Specifically, we treat H as an external entity $\mathcal{A}$ needs to query to compute H outputs, simulator SIM and reduction $\mathcal{R}$ intercept $\mathcal{A}$'s calls to H, and we measure probabilities $p_0 = \Pr[1 \leftarrow \mathbf{Exp}^{\mathsf{real}}(\mathcal{A}, \ell)]$ and $p_1 = \Pr[1 \leftarrow \mathbf{Exp}^{\mathsf{ideal}}(\mathcal{A}, \mathsf{SIM}, \ell)]$ over the randomness of H. For simplicity of notation we assume that group G is fixed for every security parameter ℓ and we assume a non-uniform security model both for the OMDH-IO assumption and UOKMS security.

We will first describe game $\mathbf{G}$, which reproduces the same distribution $\mathcal{A}$ sees in the real security game $\mathbf{Exp}^{\mathsf{real}}$, but does it in a way which makes it easier to understand simulator SIM which we will describe next. Game $\mathbf{G}$ picks $k \in \mathbb{Z}_q$ and sets the first epoch key as $(k_0, y_0) = (k, g^k)$. Game $\mathbf{G}$ also picks a list of N random group elements $g_1, ..., g_N$ in G. Then for every $i > 0$, $\mathbf{G}$ picks the following values: If $\mathcal{A}$ corrupts the server in epoch i, via a call to CorSrv, then $\mathbf{G}$ picks random $k_i \leftarrow \mathbb{Z}_q$ and outputs (k_i, y_i) for $y_i \leftarrow g^{k_i}$. (If $\mathcal{A}$ corrupts the server for two epochs in the row $\mathbf{G}$ also outputs $\Delta_i = k_{i-1}/k_i$.) If $\mathcal{A}$ corrupt a client in epoch i, via a call to CorClt, then $\mathbf{G}$ acts depending on which party $\mathcal{A}$ corrupted in epoch $i-1$: (case 1) If it was the client then $\mathbf{G}$ picks random $\Delta_i \leftarrow \mathbb{Z}_q$ and outputs (y_i, C', Δ_i) for $C' \leftarrow \{(h^{\Delta_i}, e)\,|, (h, e) \in C\}$ and $y_i \leftarrow {y_{i-1}}^{1/\Delta_i}$; (case 2) If it was the server then $\mathbf{G}$ picks random $\Delta_{j+1,i} \leftarrow \mathbb{Z}_q$ and outputs (y_i, C') for $y_i \leftarrow {y_j}^{1/\Delta_{j+1,i}}$ $C' \leftarrow \{(h^{\Delta_{j+1,i}}, e)\,|, (h, e) \in C\}$, where j was the last epoch when $\mathcal{A}$ corrupted the client *before* $\mathcal{A}$ corrupted the server in epoch $i-1$. Let E_S be the set of epochs when $\mathcal{A}$ corrupts the server and E_C the set of epochs when $\mathcal{A}$ corrupts the client. The above process defines value δ_i for each $i \in E_C$ s.t. $y_i = y^{1/\delta_i}$ (hence $k_i = k/\delta_i$), and $\mathbf{G}$ can compute this δ_i as either $\delta_{i-1} \cdot \Delta_i$, if $(i-1) \in E_C$, or as $\delta_j \cdot \Delta_{j+1,i}$, if $j \in E_C$ and $\{j+1, ..., i-i\} \subseteq E_S$. Given these values, $\mathbf{G}$ services oracles Enc, Dec, and UEnc at round $i \in E_C$ (note that these calls are disallowed if $i \in E_S$) as follows:

- $\mathbf{G}$ replies to n-th call to $\mathsf{Enc}(m)$ with $c = (h, \mathsf{Enc}_{\mathsf{dek}}(m))$ where $h = (g_n)^{\delta_i}$ and $\mathsf{dek} = H(z)$ for $z = (g_n)^k$; (Note that $z = h^{k/\delta_i}$, hence c is distributed as in the real interaction.)
- $\mathbf{G}$ replies to message u to Dec with $v = (u^{1/\delta_i})^k$;
- $\mathbf{G}$ replies to $\mathsf{UEnc}(c)$ for $c = (h, e)$ with (h', e) for $h' = h^{\Delta_i}$ if $(i-1) \in E_C$, and $h' = (h^{\delta_j \cdot \Delta_{j+1,i} \cdot k_{i-1}})^{1/k}$ if $(i-1) \in E_S$.

The correctness of Enc and Dec responses follows because $k_i = k/\delta_i$, and as for UEnc, note that $k_i = k/\delta_i$ where $\delta_i = \delta_j \cdot \Delta_{j+1,i}$ and at the same time

$k_i = k_{i-1}/\Delta_i$, which together implies that $\Delta_i = \delta_j \cdot \Delta_{j+1,i} \cdot k_{i-1} \cdot (1/k)$. Thus game $\mathbf{G}$ reproduces the exact same view as security game $\mathbf{Exp}^{\mathsf{real}}$.

Simulator SIM interacts with the ideal security game $\mathbf{Exp}^{\mathsf{ideal}}$ and executes the same algorithm as game $\mathbf{G}$ in all steps – including picking the initial key k and keys k_i if $i \in E_S$ and update-related values Δ_i or $\Delta_{j+1,i}$ if $i \in E_C$ as desrcibed above (and defining corresponding δ_i's and k_i's) – except for handling of oracles Enc and Dec, which SIM does as follows. (Note that by adaptive security of SKE Enc there is a simulator SIM_E for this SKE, which simulator SIM can utilize.) When $\mathcal{A}$ sends t-th query m to oracle Enc in epoch $i \in E_C$ then SIM receives $|m|$ and replies to $\mathcal{A}$ with $c = (h, e)$ for $h = (g_t)^{\delta_i}$ and e computed by SIM_E on input $(t, |m|)$. (2) When $\mathcal{A}$ sends u to Dec, SIM replies with $v = (u^{1/\delta_i})^k$ but then monitors $\mathcal{A}$'s queries to H: If $\mathcal{A}$ makes query z to H s.t. $z^{1/k} = g_t$ for $g_t \in \{g_1, ..., g_N\}$ then SIM asks $\mathbf{Exp}^{\mathsf{ideal}}$ to reveal message m at the t-th position in list L, and then sends (t, m) as the Reveal query to SIM_E, gets key dek as SIM_E's response, and defines $H(z) = \mathsf{dek}$. By the adaptive security of the SKE, pair (dek, e) produced by SIM_E is indistinguishable from random dek and $e = \mathsf{Enc}_{\mathsf{dek}}(m)$, in particular this process sets $H(z)$ to a value indistinguishable from random.

The only difference between $\mathcal{A}$'s interaction with $\mathbf{G}$ and $\mathcal{A}$'s interaction with SIM (which in turn interacts with $\mathbf{Exp}^{\mathsf{ideal}}$) is if in the latter case $\mathcal{A}$ queries H on arguments $(g_i)^k$ for more than Q elements in $\{g_1, ..., g_N\}$ where Q is the number of $\mathcal{A}$'s decryption queries: Given Q decryption queries SIM is allowed to learn only Q items in list L, so it can embed correct messages as decryptions of Q challenge ciphertexts, involving Q challenge points $\{g_{j_s}\}_{s=1,...,Q}$, but SIM will not be able to decrypt correctly the $(Q+1)$-st ciphertext (h, e) formed as $h = (g_{jQ+1})^{\delta_i}$ s.t. $\mathcal{A}$ queries H on $z = (g_{jQ+1})^k = h^{k_i}$. In other words, if there is ϵ difference between $Pr[1 \leftarrow \mathbf{Exp}^{\mathsf{real}}(\mathcal{A}, \ell)]$ and $\Pr[1 \leftarrow \mathbf{Exp}^{\mathsf{ideal}}(\mathcal{A}, \mathsf{SIM}, \ell)]$ then ϵ is upper-bounded by the probability that $\mathcal{A}$ queries H on values $(g_j)^k$ for $Q+1$ points g_j in $\{g_1, ..., g_N\}$. But by inspection of SIM one can see that SIM can be readily changed to reduction $\mathcal{R}$ against the OMDH-IO problem: $\mathcal{R}$ follows the algorithm of SIM except that uses the OMDH-IO challenge key g^k as y, it gets points $(g_1, \ldots, g_N)$ as part of the OMDH-IO challenge, and it uses OMDH-IO oracles $(\cdot)^k, (\cdot)^{1/k}$ instead of using exponent k directly. Note that SIM uses $(\cdot)^k$ only Q times, to service the Q decryption oracle queries, and if $\mathcal{A}$ makes queries to H on $Q+1$ arguments $(g_j)^k$ with probability ϵ, then $\mathcal{R}$ will break OMDH-IO with probability ϵ because $\mathcal{R}$ can identify such queries with oracle $(\cdot)^{1/k}$. This completes the proof of Theorem 2.

D Protocols for multiplying shared secrets

Section 6.2 requires the use of protocols for generating shares of the product of two values that are shared using Shamir's secret sharing. We recall such a protocol here.

Let $\mathsf{ProdShare}(a, b)$ be a protocol (or functionality) with n participants (referred to as servers) $S_1, \ldots, S_n$ who hold shares of secrets $a, b \in \mathbb{Z}_q$ shared

with (n, t)-threshold Shamir secret-sharing. Namely, each S_i holds a_i, b_i where $a_i = P(\alpha_i), b_i = Q(\alpha_i)$, where P (resp., Q) is a random polynomial of degree t over $\mathbb{Z}_q$ with free coefficient a (resp., b), and $\alpha_1, \ldots, \alpha_n$ are n evaluation points in $\mathbb{Z}_q$. The output of the protocol is a fresh sharing of $a \cdot b$, i.e. shares $c_1, \ldots, c_n$ where $c_i = p(\alpha_i)$ for a random polynomial p of degree t over $\mathbb{Z}_q$ s.t. $p(0) = c = a \cdot b$.

Efficient implementations of ProdShare are presented by Gennaro et al. [24]. For reference, we present a variant of the protocol for honest-but-curious servers in Fig. 14. To withstand malicious servers the protocol is amended by one or more rounds of verifiable secret sharing (VSS), i.e. it can be replaced by any of the protocols in Figures 3, 5, 6 of [24] (which differ in performance tradeoffs).

Parameters. $n \geq 2t + 1$, n distinct elements $\{\alpha_i\}_{i \in [n]}$ in $\mathbb{Z}_q$.
Server inputs. Each server S_i holds $(a_i, b_i) = (P(\alpha_i), Q(\alpha_i))$, for t-degree polynomials P, Q over $\mathbb{Z}_q$ s.t. $(a, b) = (P(0), Q(0))$.

Protocol.

1. Let $T \subseteq [n]$ be any subset of $2t + 1$ indices. For all $j \in T$, S_j chooses a random polynomial R_j of degree t over $\mathbb{Z}_q$ s.t. $R_j(0) = a_j \cdot b_j$, and for each $i \in [n]$ (including $i = j$), S_j sends $r_{ji} = R_j(\alpha_i)$ to S_i.
2. Each server S_i for $i \in [n]$ computes its share c_i of $c = a \cdot b$ as $c_i = \sum_{j \in T} \lambda_j r_{ji}$, where $\{\lambda_j\}_{j \in T}$ are Lagrange coefficients s.t. $f(0) = \sum_{j \in T} \lambda_j f(\alpha_j)$ for any $2t$-degree polynomial f.

Note that $f(x) = P(x) \cdot Q(x)$ is a $2t$-degree polynomial s.t. $f(0) = a \cdot b$ and $f(\alpha_j) = a_j \cdot b_j$, hence $c = \sum_{j \in T} \lambda_j R_j(0)$, and by linearity of Shamir secret-sharing, $p(x) = \sum_{j \in T} \lambda_j R_j(x)$ is a t-degree polynomial s.t. $c = p(0)$ and $c_i = p(\alpha_i)$ for each i.

Fig. 14. Honest-but-Curious Protocol ProdShare [24]