Updatable Oblivious Key Management for Storage Systems*

Stanislaw Jarecki
University of California, Irvine

Hugo Krawczyk
Algorand Foundation

Jason Resch
Independent

November 3, 2019

Abstract

We introduce Oblivious Key Management Systems (KMS) as a more secure alternative to traditional wrapping-based KMS that form the backbone of key management in large-scale data storage deployments. The new system, that builds on Oblivious Pseudorandom Functions (OPRF), hides keys and object identifiers from the KMS, offers unconditional security for key transport, provides key verifiability, reduces storage, and more. Further, we show how to provide all these features in a distributed threshold implementation that enhances protection against server compromise.

We extend this system with *updatable encryption* capability that supports key updates (known as key rotation) so that upon the periodic change of OPRF keys by the KMS server, a very 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. This enhances security with forward and post-compromise security, namely, security against future and past compromises, respectively, of the client's OPRF keys held by the KMS. Additionally, and in contrast to traditional KMS, our solution supports public key encryption and dispenses with any interaction with the KMS for data encryption (only decryption by the client requires such communication).

Our solutions build on recent work on updatable encryption but with significant enhancements applicable to the remote KMS setting. In addition to the critical security improvements, our designs are highly efficient and ready for use in practice. We report on experimental implementation and performance.

1 Introduction

The ever expanding cloud storage infrastructure is one of the pillars of modern computing. Yet, the key management systems (KMS) provisioning keys for the protection of the stored data have not changed fundamentally in decades. This setting involves three separate parties: a *client* $\mathcal{C}$, a remote *storage server* StS (e.g., a cloud service) that stores client data in encrypted form, and a *key management server* KmS that stores cryptographic keys for the client. The client uses the services of KmS each time it needs to encrypt or decrypt the data. The idea is that KmS is better equipped to keep keys secret and StS is better equipped to store large amounts of data reliably. Thus, KmS is charged with protecting secrecy and StS with protecting availability.

The typical deployment of such systems in practice (including large cloud-based operations such as AWS [2], Microsoft [40], IBM [27], Google [24]) uses the traditional *wrap-unwrap approach* for managing data encryption keys (dek) as shown in Fig. 1. When client $\mathcal{C}$ needs to encrypt a data object, it chooses a symmetric key dek with which it encrypts the object, then sends dek to key

*This is a full version of [32] that appeared in the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19). A preliminary treatment of the material in this paper appeared in `https://eprint.iacr.org/2018/733`.

Parties: key management server KmS, storage server StS, client $\mathcal{C}$ (= data owner).

Functions: Symmetric authenticated encryption scheme Enc;
wrapping functions Wrap, Unwrap (used to to encrypt/decrypt data encryption keys).

Keys: KmS stores a client-specific wrapping key k_c for each client.

Encryption of object $(\mathsf{ObjId}, \mathsf{Obj})$ **by client** $\mathcal{C}$**:**

1. $\mathcal{C}$ chooses random Enc key dek (data encryption key);
2. $\mathcal{C}$ sends $(\mathsf{ObjId}, \mathsf{dek})$ to KmS;
3. KmS returns $(\mathsf{ObjId}, \mathsf{wrap} = \mathsf{Wrap}_{k_c}(\mathsf{dek}))$
   (Note: KmS authenticates $\mathcal{C}$ before using k_c);
4. $\mathcal{C}$ sends $(\mathsf{ObjId}, \mathsf{wrap}, \mathsf{Enc}_{\mathsf{dek}}(\mathsf{Obj}))$ to StS for storage.

Decryption of object ObjId **by** $\mathcal{C}$**:**

1. $\mathcal{C}$ retrieves $(\mathsf{ObjId}, \mathsf{wrap}, \mathsf{Enc}_{\mathsf{dek}}(\mathsf{Obj}))$ from StS;
2. $\mathcal{C}$ sends $(\mathsf{ObjId}, \mathsf{wrap})$ to KmS;
3. KmS returns $(\mathsf{ObjId}, \mathsf{dek} = \mathsf{Unwrap}_{k_c}(\mathsf{wrap}))$
4. $\mathcal{C}$ decrypts $\mathsf{Enc}_{\mathsf{dek}}(\mathsf{Obj}))$ using dek.

Figure 1: Traditional Wrapping-based Key Management

management server KmS who *wraps* (i.e., encrypts) dek under a client-specific (master) key k_c stored at KmS and returns the result, called a wrap, to $\mathcal{C}$. Finally, $\mathcal{C}$ stores wrap and the data encrypted under dek at the storage server StS. When $\mathcal{C}$ needs to retrieve an object, it gets the corresponding ciphertext from StS, sends the attached wrap to KmS who unwraps (i.e., decrypts) it using k_c and sends dek back to $\mathcal{C}$, who uses it for decryption.

This key encapsulation mechanism, while effective and widely deployed, presents significant potential vulnerabilities. First, encryption keys dek are exposed in the clear to KmS. Second, the security of dek, hence the security of all encrypted data, relies on the channel between the client and KmS. Such a channel, typically implemented by TLS, is vulnerable to a large class of attacks, from implementation and configuration errors to certification and man-in-the-middle attacks. Third, even in normal operation, the key dek is visible to any middlebox and endpoint where TLS traffic is decrypted. Additionally, KmS can trace objects being encrypted/decrypted via the wrap values. A further shortcoming is the cost of rotating a client key by KmS: Changing the value k_c for a new $k_c{}'$ requires the client (or StS) sending *each* wrap to KmS for unwrapping under k_c and re-wrapping under $k_c{}'$. This is not only a performance issue but a security one too (due to long period of time till all wraps are updated and till k_c can be safely erased).

Oblivious KMS. Our first contribution is a simple approach to key management based on *Oblivious Pseudorandom Functions (OPRF)* [42, 22, 29], that addresses the above vulnerabilities and offers additional features absent in traditional systems. OPRFs are interactive schemes between a server holding a key to a PRF and a client holding an input. At the end of the interaction the client 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 and there are very efficient OPRF implementations, e.g. based on the Diffie-Hellman (DH) problem in regular elliptic curve groups [16, 48, 41, 26, 21] (see Fig. 3).

In our *Oblivious Key Management System (OKMS)* (see Fig. 2), a client $\mathcal{C}$ who requires a data encryption key dek for encrypting a data object interacts with the OKMS server in an OPRF protocol. $\mathcal{C}$'s input is an identifier for the data object while the server's input is an OPRF key

(typically unique per client and denoted k_c), and $\mathcal{C}$ uses the output from the OPRF as the dek[1]. In this way, the OKMS server does not learn dek (or even the object identifier). The system does not rely on an external secure channel (e.g., TLS) to transport dek; instead dek is protected by the security properties of the OPRF.[2]

This addresses two major vulnerabilities of traditional KMS systems: visibility of the dek to the server and potential exposure of this key in transit between client and server. Moreover, using the most efficient DH-based implementations of OPRFs, the protection against these threats is *unconditional*. Even a computationally unbounded server (that knows the OPRF key) or a network eavesdropper cannot learn anything about the dek, or about the object identifier input into the OPRF. Note that in OKMS, the only way for an adversary to decrypt a ciphertext is by impersonating the legitimate client or by learning the OPRF key k_c and the corresponding ObjId value. In contrast, in traditional systems, data encryption keys dek are potentially vulnerable even if the KmS key is well protected (e.g., inside a hardware module) as the dek are transmitted outside the protected zone.

The OPRF approach supports additional properties that enhance security even further and beyond anything offered by the traditional solutions. First, it provides verifiability, namely, the ability of KmS to prove to $\mathcal{C}$ that the returned dek is indeed the value that results from computing the OPRF on the client-provided object identifier. This prevents data loss that occurs if the returned dek is wrong (either due to computing error or to adversarial action); indeed, encrypting data with an incorrect, or irrecoverable, key can lead to irreparable data loss. Second, the DH-based OPRF, hence also the OKMS using it, is amenable to distribution as a multi-server threshold scheme where the OPRF key is protected as long as less than a defined threshold of the servers is corrupted. Finally, the described system can be adapted to also support *updatability*, namely, periodic key rotation of the client master key k_c by KmS with a very efficient (non-interactive) procedure for updating ciphertexts to be decryptable by the new key and not by previous ones. This procedure does not endangered the secrecy of the data and therefore can be performed by the StS. The design of such system is the main technical contribution of our work and is discused next.

Updatable Oblivious KMS. Traditional wrapping-based key management systems as those described above (and in Fig. 1) require client keys k_c to be *updated periodically* by the server KmS. Such update, known as *key rotation*, is needed to limit the exposure of data upon the exposure of k_c. For traditional wrapping systems, changing k_c with a new k_c' involves *unwrapping and re-wrapping all of a client's ciphertexts* as well as transmitting all these wrap values between the storage server and KMS server. Moreover, an old key k_c cannot be erased until *all* ciphertexts are updated to the new key k_c', extending the exposure period of k_c significantly.

This need to update clients' keys in storage systems (and other applications) has led to the notion of *updatable encryption* [9] whose goal is to provide more efficient and more secure solutions to this key rotation problem. Many flavors of updatable encryption have been suggested [9, 10, 20, 37]. In this work we investigate this notion in the context of our oblivious KMS approach leading to the design of an *Updatable Oblivious KMS (UOKMS)*.

In UOKMS, upon the rotation of a client's key k_c, server KmS computes a *short update token* Δ as a function of the old and new keys k_c, k_c', and transmits Δ to client $\mathcal{C}$. Using Δ, $\mathcal{C}$'s storage server StS can transform *all* ciphertexts that were encrypted with keys derived from k_c into ciphertexts decryptable by the new k_c' but not by the old k_c. This operation preserves the security of the data, it is performed locally at the storage server StS *without any interaction with* KmS, and it only modifies a short component of the ciphertext (independent of the length of the encrypted data) making the whole operation highly efficient. Security-wise it protects against future and past

[1] Alternatively, the output of the OPRF can be used as a key-encrypting key (kek) to locally encrypt dek.

[2] A TLS connection can be used to transport auxiliary information or client credentials but is not needed for transporting data encryption keys.

compromises of the client's key k_c.

The above UOKMS scheme offers another major performance advantage compared to traditional KMS and our own OKMS scheme: Encryption of data requires *no interaction* with the KMS server, and an interaction is only needed to decrypt data. More generally, our UOKMS supports public key encryption, so everyone can encrypt data for client $\mathcal{C}$, but only $\mathcal{C}$ can decrypt it, via an interaction with the KMS server.

Threshold Updatable OKMS. Both OKMS and UOKMS solutions can be implemented via distributed servers so that clients' OPRF keys are secure for as long as no more than a threshold number of servers are compromised. These systems inherit the high efficiency of Threshold OPRF constructions [30] (also in the case of the OPRF variant used in the UOKMS solution). In the UOKMS setting, the update token Δ is computed distributively among the servers through an efficient multi-party computation. These solutions preserve the verifiability property of OPRFs and they can be implemented in a *client-transparent* way, namely, the client's operations and code are identical regardless of the implementation as a single-server or multi-server. See Section 5.

Formal model and analysis. We formally analyze our UOKMS solution in an Updatable Oblivious KMS security model that shares close similarities with recent models of updatable encryption (or encryption with *key rotation*) [9, 10, 20, 37], but also has some significant differences. One crucial difference comes from the key management setting treated here where the client interacts with *two outsourced remote services* KmS and StS. In particular, this raises potential security vulnerabilities arising from the communication channel between client and KmS. This major concern is absent from previous updatable encryption models that treat the client and KmS essentially as collocated entities. The other aspect that is unique to our solution and formal treatment is the obliviousness of computation on the side of KmS. Yet another difference is that while the typical storage setting does not require public key encryption, we naturally include this setting in our updatable model.

Our updatability model allows attacks on both KmS and StS, including exposure of client keys k_c, update values Δ, and the attacker's ability to see and write ciphertexts into StS. Security is provided against future and past attacks, namely, forward and post-corruption security, with a simulation-based security model. Obviously, the model disallows attack combinations that would lead to trivial wins for the attacker (e.g., decrypting a challenge ciphertext in a period for which it learns the KMS key k_c). The model accommodates the oblivious setting where an attacker that communicates with KmS (and is in possession of $\mathcal{C}$'s credentials) can decrypt *any* q ciphertexts after q interactions with KmS, but all other ciphertexts remain secure. This, together with the attacker's capability to access a ciphertext-update oracle and the use of authenticated encryption, achieves CCA-like security for oblivious and updatable encryption. The security proof for our UOKMS scheme, presented in Section 4, carries in the random oracle model under a strengthened variant of the Gap One-More Diffie-Hellman assumption [5, 33] that we show to hold in the generic group model.

Implementation and performance. In Section 6 we present performance information from our implementation of both OKMS and UOKMS solutions showing the practicality of our techniques, in particular the ability of servers to support a large number of operations and clients per second. In OKMS, client 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-thread and single CPU core, and server operations are only needed for unwrapping. We also demonstrate good 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 in both single-server and multi-server deployments. Finally, we discuss implementation experience managing KmS keys in Hardware Security Modules (HSM).

1.1 Comparison to previous work

We are the first to present a comprehensive updatable solution to the central problem of key management in cloud-based (and other) storage systems that exploits the power of oblivious computation, and the first to develop a security model for such setting. Our motivation and modeling bear similarities with recent models of Updatable Encryption (UE) [9, 10, 20, 37, 35], but also has some significant differences. Most prominent is the use of obliviousness as a way to address potential vulnerabilities arising from a *remote* key management system, as opposed to one that is collocated with the client as was asssumed in all the above works on updatable encryption. Other novel features of our solution include unconditional hiding of data encryption keys and object identifiers from KmS, and building a distributed UOKMS service via a threshold implementation. Our updatability solution is ciphertext-independent (namely, the update token is of size independent from the number of ciphertexts and size of data to be updated) as in several prior UE schemes [9, 20, 37, 35]. Among those, our scheme is the most efficient, requiring a single short update value Δ from the KmS server and a single exponentiation per object for the update operation, compared e.g. to two exponentiations *per ciphertext block* in the schemes of [37, 35]. Our UOKMS scheme can be extended to provide ciphertext indistinguishability and unlinkability similarly to e.g. [37, 35], but it would inherit the inefficiency of such solution making it impractical in any large-scale data storage deployment.[3] Finally, our model and solution are the first to support public key encryption, including CCA-like security in the setting of oblivious encryption. We elaborate further on the relation to prior updatable encryption work in Section 3.2.

Updatable encryption is closely related to *proxy re-encryption (PRE)*, in particular, the Diffie-Hellman techniques at the center of our implementation directly relate to the PRE scheme of Blaze et al. [7]. Recently, [18, 17] treat forward secrecy and post-corruption security in the context of PRE for which they define evolutionary keys as in our context. However, the requirements of PRE, particularly as set forth in [17], are more stringent than needed in our case. These include generating update values using the delegatee's public key rather than on input its secret key, achieving unidirectionality, supporting general DAG delegation graphs, ensuring ciphertext indistinguishability, and more. As a result, they require more involved and less efficient techniques; in particular, [18] builds on pairing-based constructions and HIBE [14, 8] while [17] uses lattice-based fully-homomorphic techniques from [11, 47]. On the other hand, in spite of their stronger properties, none of these schemes support oblivious computation.

Our use of OPRF function can be seen as an "OPRF-as-a-service" application, a term coined in [19]. We borrow the notion of updatable oblivious PRF from that work, but their application was targeted to password verification protocols, while ours is a general encrypted storage system. (Moreover, the protocol of [19] is significantly less efficent as it uses groups with bilinear maps to obtain the stronger notion of updatable "partially oblivious" PRF, which we do not require.) OPRF's are also used in "password-protected secret sharing" [28] which can implement distributed password-secured storage but without the ability to update the master encryption key. Moreover, both of these solutions are specialized for password-authenticated clients while UOKMS accommodates any client-to-KMS or client-to-StS authentication mechanisms.

Comparison to U-PHE. The goals of UOKMS bear some similarity to Updatable Password-Hardened Encryption (U-PHE) of [36]. In the U-PHE setting a server S stores encrypted data on behalf of its clients. The encryption and decryption of data *require* S to hold the client's password and involve an interaction of S with an additional server R, called the *rate limiter*. In particular, an attacker who learns S's state *(but not the stored client password)*, cannot decrypt client's data without guessing

[3] Several prior works, e.g. [37, 35], consider ciphertext unlinkability (over update periods) as a major design goal, but achieve it at the cost of requiring $O(n)$ exponentiations to update a ciphertext of length n. We believe that in most practical settings, linkability would still be possible via metadata, object identifiers, etc., hence not worth the high computational cost it entails.

Functions: OPRF F and symmetric authenticated encryption scheme Enc.

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

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

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

Verification of correct computation of dek: Use a verifiable OPRF [28].

Figure 2: Oblivious KMS (OKMS)

the client's password and interacting with the rate limiter R. The solution offers verifiability and updatability similarly to our case, and in terms of our UOKMS model one can think of S as the storage server StS and R as the key management server KmS. However, in contrast to UOKMS, in U-PHE the server S learns both the client's decrypted message and the client's password (in particular, one relies on TLS for transmitting the password), while in UOKMS only the client encrypts and decrypts data and neither server learns it. Moreover, the U-PHE decryption protocol is not oblivious, i.e. server R, i.e., KmS, can identify the decrypted ciphertext. Also, as in the case of [19, 28] above, PHE is specialized to the password authentication case, while UOKMS is independent of the means of authentication used by clients, allowing any form of client authentication credentials. Additionally, the U-PHE scheme of [36] is less efficient than our UOKMS, specifically their encryption is interactive while ours is not, their decryption and update are both roughly twice more expensive than ours, and a threshold implementation of the rate-limiter server of [36] would be significantly more expensive than our threshold KmS.

2 Updatable Oblivious KMS

We present our main scheme, UOKMS (for Updatable Oblivious KMS), that builds on the general approach to Oblivious KMS described in the introduction and recalled next.

2.1 Oblivious Key Management System

Figure 2 specifies the Oblivious KMS (OKMS) protocol that serves as a basis for our Updatable scheme in the next section. OKMS is described and motivated in the Introduction as a much more secure alternative to the wrapping-based approach (Fig. 1) in wide use today in storage systems, particularly in large cloud deployments. When implemented with the DH-based OPRF scheme $\mathsf{dh\text{-}op}$ from Fig. 3, one obtains an OKMS that is highly efficient (see Sec. 6) and accommodates extensions to verifiability and distributed implementation (Sec. 5). The security of the OKMS scheme and its implementation using $\mathsf{dh\text{-}op}$ follows from the OPRF properties (in particular as studied in [28, 29]). We do not formally analyze the OKMS scheme but rather do so in Sections 3 and 4 for its extension to the Updatable OKMS setting presented next. (A model and analysis of OKMS can be obtained by specializing the UOKMS model to a single update period.)

Components: G: group of prime order q; H, H': hash functions with ranges $\{0,1\}^\ell$ and G, respectively, where ℓ is a security parameter.

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 client $\mathcal{C}$ and server S

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

Figure 3: DH-based OPRF function dh-op [29]

Setting: Generator g of group G of prime order q; symmetric authenticated encryption scheme $\mathsf{Enc}, \mathsf{Dec}$ with keys of length security parameter ℓ; hash function $H : G \to \{0,1\}^\ell$.

Client keys: KMS server KmS stores a client-specific random key $k_c \in \mathbb{Z}_q$ for each client; storage server StS stores certified public value $y_c = g^{k_c}$ for client $\mathcal{C}$.

Encryption of object Obj: To encrypt Obj under key y_c, pick $r \leftarrow_R \mathbb{Z}_q \setminus \{0\}$, set $w = g^r$ and $\mathsf{dek} = H({y_c}^r)$, and output ciphertext triple $c = (\mathsf{ObjId}, w, \mathsf{Enc}_{\mathsf{dek}}(\mathsf{Obj}))$.

Decryption of ciphertext $c = (\mathsf{ObjId}, w, e)$: (1) $\mathcal{C}$ checks that w is *valid*, i.e., $w \in G \setminus \{1\}$, and aborts if not; (2) $\mathcal{C}$ sends $u = w^{r'}$ for $r' \leftarrow_R \mathbb{Z}_q$ to KmS; (3) KmS checks if $u \in G$ and if so returns $v = u^{k_c}$ to $\mathcal{C}$;
(4) $\mathcal{C}$ outputs $\mathsf{Obj} = \mathsf{Dec}_{\mathsf{dek}}(e)$.

Key rotation and update: To change client's key from k_c to ${k_c}'$, KmS sends $\Delta = k_c/{k_c}'$ and $y'_c = g^{{k_c}'}$ to StS. StS replaces y_c with y'_c and replaces each ciphertext $c = (\mathsf{ObjId}, w, e)$ with $c' = (\mathsf{ObjId}, w' = w^\Delta, e)$, provided that $w \in G$. (Element $w \notin G$ indicates an invalid ciphertext which can be removed.)

Figure 4: Updatable Oblivious KMS Scheme

2.2 Updatable OKMS

Key management systems are required, by regulations and best practices, to periodically update client keys k_c (an operation known as *key rotation*). The goal is to limit the negative effects of the compromise of a key k_c to a shorter period of time and to as little data as possible. This is particularly important for keys that protect data stored for long periods of time as it is common in many cloud storage applications (anything from user photos to regulated financial information). Upon the rotation by the KMS server KmS of a key k_c into a new key ${k_c}'$, all ciphertexts protected with k_c and held by the storage server StS need to be updated too. The updated ciphertexts should be decryptable by ${k_c}'$ but not by k_c. The goal is that an attacker that learns k_c but only sees updated ciphertexts should not be able to learn anything about the encrypted data in the new period (while ${k_c}'$ is unexposed). Similarly if the attacker has seen a ciphertext encrypted using an unexposed k_c and later learns ${k_c}'$, it still should not learn anything from that ciphertext. This provides both *forward security* (security against future exposures) and post-compromise security (security against past exposures). Obviously, one also requires that the update process itself does

not reveal encrypted information to StS (e.g., decrypting and re-encrypting the data by StS would not be considered secure).

In the traditional wrapping-based KMS of Fig. 1, such key rotation operation requires interaction between the storage server StS and KMS server KmS where StS sends *every* stored wrap to KmS for unwrapping under k_c and re-wrapping using k_c'. This requires the transmission of all wrap values between StS and KmS, and the exposure of all dek values to KmS. In a large storage setting such process can take *very long time* (particularly under the "lazy evaluation" practice where a wrap held by StS is updated to k_c' only when the application requires a regular unwrap operation for that object). During all this time the old and new keys k_c, k_c' must be stored at KmS thus extending the life and exposure period of these keys.

In Fig. 4 we present an *Updatable Oblivious KMS* that adapts the OKMS scheme from the previous section to the updatable setting. Using techniques from *updatable encryption* [9, 20, 37] adapted to the oblivious setting, we achieve some desirable properties, both in terms of security and performance. First, upon the change of key k_c into a new key k_c', KMS server KmS can produce a *short token* Δ with which *all* the ciphertexts of client $\mathcal{C}$ can be updated by StS in a way that achieves the above security properties. Second, the update operation is *non-interactive*: It is performed locally by StS with the sole possession of Δ. Note that once KmS produces a new key k_c' and the corresponding update value Δ, KmS can immediately erase the old key k_c, hence reducing the risk of exposure to only one key at a time. Finally, the update operation at StS only requires a single exponentiation per ciphertext independently of the ciphertext size, compared to at least 2 exponenentiations per ciphertext in previous updatable encryption schemes (see also footnote 3), leading to a fast update of all ciphertexts that were encrypted under k_c. Thus, one obtains a very efficient update procedure that achieves better security than in the wrapping-based KMS in many ways: dek keys are never exposed to StS or to KmS during updates; old keys can be erased immediately upon rotation; the interaction between StS and KmS is minimal (only Δ is transmitted); and Δ can be erased by StS as soon as it locally updates all ciphertexts.

The UOKMS scheme from Fig. 4 departs from the OKMS scheme of Fig. 2 in some important ways. First, to allow for fast updates, ciphertexts are composed of two parts, a wrap and a symmetrically encrypted ciphertext that derives the encryption key from wrap. For updates, only wrap is updated. Second, the *encryption operation is non-interactive*, that is, $\mathcal{C}$ (or anyone else) can encrypt data locally without interacting with KmS provided that it possesses the equivalent of a certified "public key" y_c corresponding to k_c ($y_c = g^{k_c}$ in our scheme). Decryption is only possible via an oblivious interaction with KmS. As a "side effect" of the above properties, the UOKMS scheme supports public key encryption, meaning that anyone can produce ciphertexts but only $\mathcal{C}$ can decrypt them, thus expanding the use cases for such KMS solution. Note that decryption requires interaction with KmS which we assume has the means to authenticate decryption requests from $\mathcal{C}$. Third, the UOKMS scheme from Fig. 4 is presented in terms of a specific instantiation rather than using generic tools like the OPRF in OKMS. Indeed, the malleability properties required for the update operations are not possible with a generic OPRF (but see below about Weak OPRFs). Finally, verifiability of correct encryption by KmS is not needed in UOKMS where encryption is non-interactive, and verification of correct decryption can be done via the (symmetric) authenticated decryption operation Dec. This saves the need to verify the correct exponentiation by KmS, further improving the performance of UOKMS.

Correctness of the UOKMS scheme is easy to validate. For encryption, one sets $w = g^r$ for random r, then derives the encryption key dek from y_c^r, encrypts the data and stores w. For decryption, $\mathcal{C}$ computes w^{k_c} obliviously in interaction with KmS and derives dek from this value. This recovers the original data as $y_c^r = (g^{k_c})^r = (g^r)^{k_c} = w^{k_c}$. Regarding the update operation, if we denote by w_t and k_t the values of w and k_c, respectively, after t updates (here w_0 denotes the original value of w computed at the time of deriving dek, and k_0 denotes the client's key k_c as it existed at that time), then one can see inductively that if $w_t^{k_t} = w_0^{k_0}$ (the latter is the value from

which dek is derived), then this is also true for $t+1$, namely, $(w_{t+1})^{k_{t+1}} = (w_0)^{k_0}$. Indeed, we have that $w_{t+1} = w_t^{\Delta_{t+1}} = w_t^{k_t/k_{t+1}}$, thus $(w_{t+1})^{k_{t+1}} = (w_t^{k_t/k_{t+1}})^{k_{t+1}} = w_t^{k_t} = w_0^{k_0}$.

Security of the UOKMS scheme from Fig. 4 is proven in Section 4 based on the security model presented in Section 3.

In Section 5 we show how to distribute the KmS functionality of UOKMS through a threshold scheme which includes the distributed generation of the value Δ so only StS can learn it.

On Weak Oblivious PRF. The UOKMS scheme from Fig. 4 derives symmetric encryption keys from a function $F_k(w) = H(w^k)$ defined over elements in a group G of prime order q (where the key k is chosen at random in the set $\mathbb{Z}_q$). The function F has strong similarities with the OPRF $\mathsf{dh\text{-}op}_k(x) = H(x, (H'(x))^k)$ from Fig. 3 that we use as the basis of the OKMS scheme from Fig. 2, as well as some fundamental differences. First, the input to F is a group element (rather than an arbitrary string mapped into the group by the hash function H' in dh-op). But more importantly, knowing w^k for any value w allows to compute the function on w^t for known t. At the same time, computing F on a independently random group element is hard under CDH hence F can be modeled as a Weak PRF (as noted in [41]). In our application for UOKMS we also use the fact that F can be computed obliviously and use its homomorphic properties to support updatability. We leave as a future work item the formalization of such "oblivious Weak OPRF" function in the UC model, similarly to the treatment of OPRFs in [29]. For the purpose of our use of F in the context of UOKMS, we carry the analysis directly in a specialized UOKMS security model that we present in Section 3.

3 Security Model for Updatable Oblivious KMS

We introduce the security model for Updatable Oblivious KMS which combines the elements and advantages of oblivious computation and updatable encryption in a single model. As in updatable encryption, e.g. [9, 10, 20, 37, 35], we consider keys that evolve over *epochs*, where at the beginning of a new epoch the encryption/decryption key is replaced with a fresh key. In our case, this applies to client keys k_c held by the Key Management server KmS. The goal is to capture the security of key rotation both in the sense of forward security and post-compromise security. That is, the compromise of a client key k_c from a given epoch should not help in exposing data encrypted either at a later epoch or in a previous epoch. In the latter case, however, one needs to qualify this requirement. Suppose that a ciphertext e is generated using the key k_c from epoch t and later the key $k_c{}'$ for epoch $t' > t$ is exposed; should the data d encrypted under ciphertext e still be secure? Clearly, if the attacker $\mathcal{A}$ sees e', the updated version of ciphertext e in epoch t', then $\mathcal{A}$ can decrypt e' and obtain d. However, if $\mathcal{A}$ possesses $k_c{}'$ and e but does not have the updated e' then the security of d needs to be fully preserved.

The above illustrates the intricacies of updatable encryption models, which require careful bookkeeping of information available to the attacker: What ciphertexts it sees and when, for what epochs it obtains the secret key k_c, and for which it receives update information, etc. The goal is to prevent the attacker from learning anything that is not trivially (and unavoidably) derivable from the information it requests. In this section, we set these rules and goals through a formal model of UOKMS security, and use it in Section 4 to prove the security of our UOKMS design from Fig. 4.

3.1 Formal UOKMS Scheme

Formally, an Updatable Oblivious KMS (UOKMS) scheme is a tuple of algorithms KGen, Enc, UGen, UEnc, and a protocol Dec, intended for a KMS server KmS, a storage server StS, and a client $\mathcal{C}$, s.t.:

- KGen is a key generation algorithm, run by KmS, which on input a *security parameter* ℓ generates a public key pair $(\mathsf{sk}, \mathsf{pk})$.
- Enc is an encryption algorithm, run by any party, which on input key pk and plaintext m generates ciphertext c.
- $\mathsf{Dec} = (\mathsf{Dec.KmS}, \mathsf{Dec.C})$ is an interactive decryption protocol between a client running $\mathsf{Dec.C}(\mathsf{pk}, c)$ and KmS running $\mathsf{Dec.KmS}(\mathsf{sk}, \mathsf{pk})$, where $\mathsf{Dec.C}$ outputs m or $\perp$.
- UGen is an update generation algorithm, run by KmS, which on input $(\mathsf{sk}, \mathsf{pk})$ generates a new key pair $(\mathsf{sk}', \mathsf{pk}')$ together with an *update token* Δ.
- UEnc is a ciphertext update algorithm, run by StS, which on input (c, pk, Δ) outputs an updated ciphertext c'.

An UOKMS scheme must satisfy the following *correctness* property. First, the interactive decryption must recover the encrypted plaintext, i.e. for any m, if $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}(\ell)$ and $c \leftarrow \mathsf{Enc}(\mathsf{pk}, c)$ then $\mathsf{Dec.C}(\mathsf{pk}, c)$ outputs m after an interaction with $\mathsf{Dec.KmS}(\mathsf{sk}, \mathsf{pk})$. Furthermore, the same correctness property applies to keys and ciphertexts produced and updated in later periods. That is, for every m, if $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KGen}(\ell)$, $c \leftarrow \mathsf{Enc}(\mathsf{pk}, m)$, $(\mathsf{sk}', \mathsf{pk}', \Delta) \leftarrow \mathsf{UGen}(\mathsf{sk}, \mathsf{pk})$, and $c' \leftarrow \mathsf{UEnc}(c, \mathsf{pk}, \Delta)$, then $\mathsf{Dec.C}(\mathsf{pk}', c')$ outputs m after an interaction with $\mathsf{Dec.KmS}(\mathsf{sk}', \mathsf{pk}')$.

On public and private values. We model UOKMS as a *public key* encryption scheme where *any party* in possession of the public key pk can encrypt files for the client whose corresponding decryption key sk is held by KmS. It is assumed that KmS has the means to authenticate the client before engaging in a decryption operation using key sk but a secret channel between client and KmS is not needed. The update token Δ is assumed to be transmitted from KmS to StS over a secure channel. No other party needs or should know this value. In particular, the model does not guarantee secrecy of sk_{t+1} given sk_t and Δ_{t+1} or secrecy of sk_t given sk_{t+1} and Δ_{t+1}. For example, in our UOKMS scheme of Figure 4 receiving Δ_{t+1} allows one to derive both sk_{t+1} from sk_t, and sk_t from sk_{t+1}. In this case, if Δ_{t+1} was leaked then a KmS corrupted in epoch t would be effectively also corrupted in epoch $t+1$, and vice versa.[4]

3.2 UOKMS obliviousness and security

The definition below formalizes the notion of KMS obliviousness.

Definition 1 *We say that a UOKMS scheme is* oblivious *if for all efficient algorithms $\mathcal{A}$ the interaction of $\mathcal{A}$ 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 are valid ciphertexts of the same length and pk is a valid public key.*[5]

As noted above, defining security of UOKMS, and of updatable encryption in general, requires establishing the rules of what information the adversary is entitled to receive and when, and what constitutes a win relative to that information. In our model, time is divided into *epochs* at the beginning of which a new key pair $(\mathsf{sk}, \mathsf{pk})$ and an update token Δ are generated. For each epoch the adversary $\mathcal{A}$ receives the new public key pk, and can request to see either the new secret key

[4] This is not a necessary feature of a UOKMS scheme, i.e. one could imagine that Δ_{t+1} allows for updating ciphertexts (and the public key), but not for updating the corresponding secret key. However, all existing ciphertext-independent updatable encryption schemes, ours included, allow for updating sk given Δ.

[5] The public key pk, normally chosen by KmS, can be chosen by $\mathcal{A}$ in this definition, modeling a malicious KmS, but $\mathcal{C}$ can check some properties of the public key and the ciphertext, e.g. that they contain expected group elements, before running Dec.

sk or the update token Δ, which corresponds to $\mathcal{A}$ compromising in that epoch, respectively either server KmS or server StS. Algorithm $\mathcal{A}$ is also given oracle access to the ciphertext-update function UEnc but it is not allowed to use it for trivial wins, e.g., updating challenge ciphertexts to an epoch for which it knows the secret key. Note that $\mathcal{A}$ learns the secret key sk_t of epoch t if $\mathcal{A}$ asks for it, but also if $\mathcal{A}$ asks for sk_{t-1} in epoch $t-1$ and asks for Δ_t in epoch t. This shows that what $\mathcal{A}$ can learn in one epoch may depend on what it knew in the previous epoch, and the UOKMS security game rules must reflect that.

We formalize these rules and the attacker goals via the real-ideal experiments shown in Fig. 5. In each epoch t, the attacker receives pk_t and chooses to either corrupt KmS, hence obtaining sk_t, or to corrupt StS, hence obtaining the update token Δ_t, except if KmS was corrupted in epoch $t-1$ (otherwise the attacker could calculate sk_t from sk_{t-1} and Δ_t, making this case equivalent to corrupting both KmS and StS in epoch t). In addition, $\mathcal{A}$ obtains access to oracles $\mathsf{Enc}, \mathsf{Dec}, \mathsf{UEnc}$, depending on the compromised party. An aspect of the definition that is specific to our oblivious setting is that the attacker with access to the oblivious decryption oracle can decrypt any ciphertext of its choice in a decryption call, but each call can result in decryption of at most a *single* challenge ciphertext. More generally, with q calls to the decryption oracle, $\mathcal{A}$ can decrypt q messages but nothing more. Finally, we note that the ability of the attacker to access a decryption oracle provides CCA security to our public key scheme in the oblivious setting.

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

Set $t \leftarrow 0$ and $\mathsf{corr}_0 \leftarrow \mathsf{sts}$. Generate $(\mathsf{sk}_0, \mathsf{pk}_0) \leftarrow \mathsf{KGen}(\ell)$ and give pk_0 to $\mathcal{A}$.
The experiment output is the output of $\mathcal{A}$ after interaction with the following oracles:

Enc: On $\mathcal{A}$'s input m, if $\mathsf{corr}_t = \mathsf{sts}$ output $\mathsf{Enc}(\mathsf{pk}_t, m)$;
Dec: Let $\mathcal{A}$ interact with $\mathsf{Dec.S}(\mathsf{sk}_t)$;
UEnc: On $\mathcal{A}$'s input (t', c), if $\mathsf{corr}_t = \mathsf{sts}$ and $0 \le t' < t$ then
set $c_{t'} := c$; for $j = t'+1$ to t set $c_j := \mathsf{UEnc}(c_{j-1}, \mathsf{pk}_j, \Delta_j)$; output c_t;
Corr: On $\mathcal{A}$'s input corr_{t+1}, set $(\mathsf{sk}_{t+1}, \mathsf{pk}_{t+1}, \Delta_{t+1}) \leftarrow \mathsf{UGen}(\mathsf{sk}_t, \mathsf{pk}_t)$;
If $(\mathsf{corr}_t, \mathsf{corr}_{t+1}) = (\mathsf{kms}, \mathsf{kms})$ output $(\mathsf{pk}_{t+1}, \mathsf{sk}_{t+1}, \Delta_{t+1})$;
If $(\mathsf{corr}_t, \mathsf{corr}_{t+1}) = (\mathsf{kms}, \mathsf{sts})$ output pk_{t+1};
If $(\mathsf{corr}_t, \mathsf{corr}_{t+1}) = (\mathsf{sts}, \mathsf{kms})$ output $(\mathsf{pk}_{t+1}, \mathsf{sk}_{t+1})$;
If $(\mathsf{corr}_t, \mathsf{corr}_{t+1}) = (\mathsf{sts}, \mathsf{sts})$ output $(\mathsf{pk}_{t+1}, \Delta_{t+1})$;
Increment epoch counter $t := t + 1$.

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

Set $t \leftarrow 0$ and $\mathsf{corr}_0 \leftarrow \mathsf{sts}$. Initialize an empty challenge plaintext list L.
Let a *stateful* algorithm SIM generate pk_0 on input ℓ and give pk_0 to $\mathcal{A}$.
Experiment output is the output of $\mathcal{A}$ after interaction with the following oracles:

Enc: On $\mathcal{A}$'s input m, if $\mathsf{corr}_t = \mathsf{sts}$ add m to L and output $\mathsf{SIM}(\mathsf{enc}, |m|)$;
Dec: Let $\mathcal{A}$ interact with $\mathsf{SIM}(\mathsf{dec})$ while letting SIM learn *one* chosen item in L;
UEnc: On $\mathcal{A}$'s input (t', c), if $\mathsf{corr}_t = \mathsf{sts}$ and $0 \le t' < t$ then output $\mathsf{SIM}(\mathsf{upd}, t', c)$;
Corr: On $\mathcal{A}$'s input corr_{t+1} output $\mathsf{SIM}(\mathsf{corr}_{t+1})$ and increment epoch counter $t := t + 1$.

Figure 5: Security Experiments for Updatable Oblivious KMS

Figure 5 shows two experiments: Experiment $\mathbf{Exp}^{\mathsf{real}}_{\mathsf{uokms}}(\mathcal{A}, \ell)$ which models an interaction of the real-world adversary $\mathcal{A}$ with the real UOKMS scheme, and experiment $\mathbf{Exp}^{\mathsf{ideal}}_{\mathsf{uokms}}(\mathcal{A}, \mathsf{SIM}, \ell)$ which models an interaction of a simulator SIM with an "ideal" UOKMS scheme. We call a UOKMS scheme secure if the two interactions, real and ideal, are indistinguishable. Formally:

Definition 2 *Let* $\mathbf{Adv}^{\mathsf{real}}_{\mathsf{uokms}}(\mathcal{A}, \ell)$ *be the probability that experiment* $\mathbf{Exp}^{\mathsf{real}}_{\mathsf{uokms}}(\mathcal{A}, \ell)$ *outputs* 1*, and let* $\mathbf{Adv}^{\mathsf{ideal}}_{\mathsf{uokms}}(\mathcal{A}, \mathsf{SIM}, \ell)$ *be the probability that experiment* $\mathbf{Exp}^{\mathsf{ideal}}_{\mathsf{uokms}}(\mathcal{A}, \mathsf{SIM}, \ell)$ *outputs* 1*. We say that UOKMS scheme is* secure *if for all efficient algorithms* $\mathcal{A}$ *there exist an efficient algorithm* SIM *s.t.* $|\mathbf{Adv}^{\mathsf{real}}_{\mathsf{uokms}}(\mathcal{A}, \ell) - \mathbf{Adv}^{\mathsf{ideal}}_{\mathsf{uokms}}(\mathcal{A}, \mathsf{SIM}, \ell)|$ *is negligible in* ℓ*.*

The real experiment $\mathbf{Exp}^{\mathsf{real}}_{\mathsf{uokms}}(\mathcal{A}, \ell)$ in Figure 5 models an interaction of adversary $\mathcal{A}$ with a UOKMS scheme which progresses through epochs $t = 0, 1, ...$ where the flag corr_t designates whether $\mathcal{A}$ corrupts KmS (kms) or the storage server StS (sts) in epoch t. After the initialization which generates the initial KMS key pair $(\mathsf{sk}_0, \mathsf{pk}_0)$ we give pk_0 to $\mathcal{A}$ and let $\mathcal{A}$ interact with the encryption, decryption, and ciphertext update oracles. We model the progress from one epoch to the next via "party corruption" oracle Corr which uses $\mathcal{A}$'s decision bit corr_{t+1} to corrupt either KmS or StS in the next epoch[6]. This oracle triggers a key update, i.e. a new KMS key pair is created as $(\mathsf{sk}_{t+1}, \mathsf{pk}_{t+1}, \Delta_{t+1}) \leftarrow \mathsf{UGen}(\mathsf{sk}_t, \mathsf{pk}_t)$, and the epoch counter t is incremented. Adversary $\mathcal{A}$ then receives the new public key pk_{t+1} and possibly more, depending on the parties it corrupts: Namely, if $\mathsf{corr}_{t+1} = \mathsf{kms}$ then $\mathcal{A}$ also gets the new secret key sk_{t+1}, and if $\mathsf{corr}_{t+1} = \mathsf{corr}_t$, i.e. if $\mathcal{A}$ corrupts the same party in the two consecutive epochs, then $\mathcal{A}$ also gets the update token Δ_{t+1}. Crucially, $\mathcal{A}$ does *not* get Δ_{t+1} if $\mathsf{corr}_{t+1} \neq \mathsf{corr}_t$. (Indeed, as mentioned above, in the UOKMS scheme of Figure 4, receiving the update token would allow the adversary to effectively extend the corruption of KmS from epoch t to epoch $t{+}1$ and vice versa.)

The security experiment assumes that KmS corruptions are passive in the sense that if $\mathsf{corr}_t = \mathsf{kms}$ we let $\mathcal{A}$ learn sk_t (and Δ_t if $\mathsf{corr}_{t-1} = \mathsf{kms}$), but we do not let $\mathcal{A}$ interfere in the update generation and/or the dissemination of the created update token and a public key, or in the execution of the decryption protocol. (All existing Updatable Encryption security notions make such choices, e.g. assuming that even if the adversary compromises the entity that stores the key, the key update is still generated honestly.)

We assume that StS corruptions are active in the sense that for epochs where $\mathsf{corr}_t = \mathsf{sts}$ we not only let $\mathcal{A}$ learn Δ_t, but we also give $\mathcal{A}$ an access to the ciphertext update oracle UEnc which on input (t', c), for $t'{<}t$ and c a ciphertext from epoch t', outputs the updated value of c at epoch t. That is, the oracle runs the update algorithm on (supposed) ciphertext $c_{t'} = c$ using update tokens $\Delta_{t'+1}, ..., \Delta_t$, and outputs the updated ciphertext c_t. This models the ability of the adversary to inject ciphertexts to StS in some epoch – either by directly modifying these ciphertexts when StS is corrupted, or by sending a ciphertext to the client who then stores it at StS[7] – and then having this ciphertext updated by the UEnc oracle. Note that the Update protocol is not provided at epochs where $\mathsf{corr}_t = \mathsf{kms}$ since this would allow $\mathcal{A}$ to decrypt challenge ciphertexts using the compromised KmS key.

Our UOKMS models a public key encryption where adversary $\mathcal{A}$ can encrypt any message at will, but the role of the encryption oracle Enc in the UOKMS security game is to model the generation of *challenge ciphertexts*. Namely, in the real game, oracle Enc on $\mathcal{A}$'s input m generates a ciphertext $c = \mathsf{Enc}(\mathsf{pk}_t, m)$, but in the ideal game the same ciphertext c must be produced by the

[6] We assume w.l.o.g. that $\mathcal{A}$ corrupts exactly one of these parties in each epoch. In particular, the real-world event when both StS and KmS are corrupted in epoch t is reflected in our model by KmS corrupted in two consecutive epochs $t{-}1, t$, because this reveals both sk_t and Δ_t to $\mathcal{A}$. On the other hand, the epoch without corruption strictly weakens the adversary capabilities hence it is subsumed by the other cases.

[7] Note that our treatment is of a public key encryption, so other parties can potentially create ciphertexts which land in the StS storage.

simulator algorithm SIM given only $|m|$ (and flag enc) as an input while the plaintext m is added to the (secret) list L of encrypted challenge plaintexts. Adversary $\mathcal{A}$ can decrypt any ciphertexts (or indeed any ciphertext-like objects of its choice) using the decryption oracle Dec. Because we aim to support *oblivious* decryption, the precise ciphertext which $\mathcal{A}$ effectively enters into the decryption oracle is hidden from the oracle, hence we must count each decryption oracle access as an attempt to decrypt some challenge ciphertext. We model this in the ideal game by giving SIM access to a *single* location in list L of challenge plaintexts, per each Dec query of $\mathcal{A}$. Note that this technically implies that the simulator can extract the unique ciphertext which $\mathcal{A}$ attempts to decrypt in this oblivious decryption protocol instance, or otherwise the simulator wouldn't know which plaintext on list L it should access. Observe also that we do not create challenge ciphertexts in an epoch where KmS is corrupted, because knowledge of KmS's private key makes all such ciphertexts insecure.

We stress that the $\mathbf{Exp}^{\mathsf{real}}_{\mathsf{uokms}}$ security game allows any pattern of corruptions *except* corruption of both StS and KmS in a single epoch (see footnote 6). 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 below.)

Prior Updatable Encryption Models. Our notion of Updatable (Oblivious) KMS is related to *Updatable* Encryption (UE) or Encryption with *Key Rotation*, which was studied in several recent works [9, 10, 20, 37, 35]. UOKMS extends the notion of UE by splitting the UE's client into two separate entities, the KMS server, which holds the client's decryption key and generates key updates, and the client itself, who decrypts the ciphertexts retrieved from the storage server via an interactive decryption protocol with the KMS. The UOKMS model thus lifts the notion of Updatable Encryption to the setting that reflects realistic large cloud storage deployments, where the decryption keys of all clients are held by a specialized Key Managment server. On the other hand, collapsing the client and the KMS in the UOKMS model into a single entity gives exactly the setting of UE, hence our UOKMS scheme and security notion give rise to the corresponding UE scheme and notion.

Our security model corresponds to the *ciphertext-independent* UE model of Lehmann et al. [37] (which refines the model of Everspaugh et al. [20]), where a single update message can be used to update any number of ciphertexts. Of these only the recent work of Klooss et al. [35] addresses CCA security, and lets the adversary access a decryption oracle, as we do in our model. However, of the two schemes shown secure in [35] the one whose efficiency is comparable to ours does not allow the adversary an unrestricted access to the Ciphertext Update oracle, while our model allows unrestricted access to *both* Dec and UEnc oracles. The scheme of [35] which allows such unrestricted oracle access relies heavily on pairing-based NIZKs, using e.g. 22 pairings in decryption, in contrast to a single standard group exponentiation used in decryption in our scheme. However, our model of UOKMS security is specialized to the case of oblivious interactive decryption where the decryption oracle, which models the KMS server, runs on *blinded* ciphertexts. In such setting a standard CCA notion, where the decryption oracle is restricted from decrypting a challenge ciphertext, does not apply. Thus we capture security with a "counting method" which enforces that any Q accesses to the decryption oracle allow for learning plaintext information in at most Q challenge ciphertexts. Ours is the first treatment of Updatable Encryption with *oblivious* decryption procedure, and this setting necessitates a "counting-based" notion of security in the presence of decryption oracle.

The UE schemes of [20, 37, 35] achieve *update indistinguishability*, i.e. a ciphertext updated to the new epoch cannot be efficiently linked to the original from the previous epoch. We do not consider this property, although our scheme can be extended to support it, because achieving this property requires update cost proportional to the *total size* of the encrypted data, which we believe is impractical in large storage deployments (see footnote 3). The above UE schemes also consider *ciphertext integrity*, but this notion is specialized to the case of symmetric key encryption, while our UOKMS model treats the case of public key encryption.

Finally, we should point out that our security model is static in the sense that an adversary must choose at the beginning of each epoch whether it compromises the decryption key stored by the KMS or the update token held by the storage server (or both). By contrast, [37, 35] consider an adaptive model of corruptions, where an adversary can request either the decryption key or the update token or both for any *past* epoch as well. The adaptive security model is more general and less restrictive, but we analyze the security of our scheme only in the static model because adaptive security presents subtle technical challenges which we do not know how to overcome.[8] Technically, the simulator would have to make bets about past epochs, guessing whether an adverary will eventually ask for a decryption key for some past epoch (in which case the simulator needs to know this epoch key), or whether an adversary will ask for an update token which allows updating a challenge ciphertext to that epoch (in which case the simulator needs to embed an encryption challenge in that epoch key). Since the simulator needs to make these bets with respect to polynomially-many past epochs, the probability that its guesses are all correct will be negligible, and it is not clear if such strategy can lead to efficient simulation. We thus believe that security analysis in the fully adaptive model of [37, 35] remains an open question.

4 Security Analysis of the UOKMS Scheme

The UOKMS scheme shown in Figure 4 is information-theoretic *oblivious*, as is the OPRF protocol dh-op on which the Decryption protocol in Fig. 4 is based, but the *security* of this scheme relies on the *OMDH-IO* computational assumption and the *(receiver) non-committing* property of symmetric encryption, both defined below:

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 $\mathcal{A}$'s queries to the $(\cdot)^k$ oracle.

Without access to oracle $(\cdot)^{1/k}$, the above is identical to the *One-More DH* (OMDH) assumption [6, 33], which was used e.g. for proving the security of the practical OPRF schemes [28, 29], particularly the one shown in Figure 3 in Section 2.1. Thus, OMDH-IO is a strengthening of OMDH; its security can be proven in the Generic Group Model (GGM) as an extension to the proof of OMDH in that model [31] and with a slight modification of the security bounds. We sketch this adaptation in Appendix A.

Receiver Non-Committing Symmetric-Key Encryption. This property of symmetric-key encryption (SKE) is used in our security analysis to enable the simulation required by the security game in Fig. 5. Informally, it states that without knowledge of the encryption key, ciphertexts do not commit to their underlying plaintexts, thus allowing the simulator to "explain" a fixed ciphertext as the encrypton of any plaintext. Formally, a symmetric encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ is *receiver non-committing* (RNC) if for any PPT $\mathcal{A}$ there exists PPT SIM s.t. $\mathcal{A}$'s view in the following real and ideal games is indistinguishable: (1) In the real game $\mathcal{A}$ interacts with oracles Enc and Reveal, where $\mathsf{Enc}(i, m)$ picks random key k_i and outputs $e = \mathsf{Enc}(k_i, m)$ while $\mathsf{Reveal}(i)$ reveals k_i; (2) In the ideal game $\mathcal{A}$ interacts with a *stateful* algorithm SIM, s.t. when $\mathcal{A}$ sends (i, m) as an Enc query, SIM must return e on input $(i, |m|)$, and when $\mathcal{A}$ sends i as a Reveal query, SIM must output k_i on input (i, m).

[8] We stress that this is an issue in the proof only and not an explicit attack, and that similar technical issues were observed regarding adaptive security in other contexts, e.g. in proactive cryptosystems, see e.g. [13, 1, 38].

Theorem 3 *The UOKMS scheme in Figure 4 is unconditionally* oblivious *and is* secure *under the OMDH-IO assumption in ROM if the symmetric encryption scheme* Enc *is receiver non-committing.*

Notes on the Proof. The proof of Theorem 3 is presented in Section 4.1. We note that the inverse exponentiation oracle in OMDH-IO is necessary to obtain the theorem as the protocol (in the context of our model) provides an attacker $\mathcal{A}$ that corrupts KmS in epoch $t-1$ and StS in period t with an oracle to the function $(\cdot)^{1/k_t}$. Indeed, in epoch t, $\mathcal{A}$ obtains access to UEnc which implements an exponentiation oracle $(\cdot)^{\Delta_t} = (\cdot)^{k_{t-1}/k_t}$, and together with knowledge of k_{t-1}, $\mathcal{A}$ can compute $(\cdot)^{1/k_t}$ on any value of its choice. The RNC property of SKE Enc is likewise necessary. Consider an attacker $\mathcal{A}$ making two queries: (a) an Enc query on some m, and (b) a Dec query where $\mathcal{A}$ runs the Dec.C protocol on the received ciphertext $c = (\mathsf{ObjId}, w, e)$. By the UOKMS security game rules of Fig. 5 the simulator SIM has to simulate this as follows: (a) it produces c on message length $|m|$, and (b) on input m retrieved from list L, it simulates protocol Dec.S so that c decrypts to m. SIM's response v to $\mathcal{A}$'s message u in the decryption protocol, see Fig. 4, defines the effective KMS key as $k = \mathsf{DL}(u, v)$, and consequently defines the data encryption key for (ObjId, w, e) as $\mathsf{dek} = H(w^k)$. Thus when SIM defines the output dek of oracle H on input w^k it must satisfy that $e = \mathsf{Enc}_{\mathsf{dek}}(m)$. In particular, SIM first creates ciphertext e given just $|m|$ and then, given m, it creates dek s.t. $e = \mathsf{Enc}_{\mathsf{dek}}(m)$, wihch implies that SKE Enc satisfies the RNC property.

Corollary 4 *The UOKMS scheme in Figure 4 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.*

The corollary follows because CTR and CBC encryption modes satisfy the receiver non-committing property in the Ideal Cipher model: If message length $|m|$ defines n blocks for block cipher E then SIM services Enc query on input $(i, |m|)$ by setting ciphertext $e = (IV, e_1, ..., e_n)$ where IV and all e_i's are random blocks. When SIM gets $m = (m_1, ..., m_n)$ to service Reveal(i) query, values $(IV, e_1, ..., e_n, m_1, ..., m_n)$ define n input/output pairs which SIM needs to set for $E(k, \cdot)$ for random key k. For counter mode CTR, SIM sets $E(k, IV + j) = m_j \oplus e_j$ for all j while for CBC mode, SIM sets $E(k, m_j \oplus e_{j-1}) = e_j$ for all j where $e_0 = IV$. Either way by randomness of e_i's this sets $E(k, \cdot)$ outputs on n given points to n random values. The probability that this creates collisions in $E(k, \cdot)$ is negligible, and by randomness of k there is a negligible probability that any points of $E(k, \cdot)$ were queried before.

Note. The above argument can be expanded to include authenticated encryption via encrypt-then-mac where the simulator chooses the MAC key.

4.1 Proof of Theorem 3

Proof: Note first that the *unconditional obliviousness* of this UOKMS scheme is immediate, because for any public key pk and any two valid ciphertexts $c_0 = (\mathsf{ObjId}_0, w_0, e_0)$ and $c_1 = (\mathsf{ObjId}_1, w_1, e_1)$, the interaction with Dec.C on (pk, c_b) for $b = 0$ and $b = 1$ is identical: In either case $\mathcal{C}$ sends $u = (w_b)^{r'}$ for $r' \leftarrow_{\mathrm{R}} \mathbb{Z}_q$, which is a random group element if $w_b \in G$ and $w_b \neq 1$ because the group order is prime. To argue UOKMS *security* we will first show an efficient simulator algorithm SIM which having access to (any) adversary algorithm $\mathcal{A}$, interacts with the ideal UOKMS game $\mathbf{Exp}^{\mathsf{ideal}}_{\mathsf{uokms}}$. 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 $\mathbf{Exp}^{\mathsf{real}}_{\mathsf{uokms}}$ and an interaction with SIM and $\mathbf{Exp}^{\mathsf{ideal}}_{\mathsf{uokms}}$, i.e. if

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

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. We note that in one step along these SIM modifications we replace the real symmetric encryption Enc with the simulator assumed by the RNC property of SKE Enc. We provide the details of the proof now.

The proof relies on the ROM model for function $H : G \to \{0,1\}^\ell$ used in UOKMS scheme in Figure 4. 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 $p_0 = \Pr[1 \leftarrow \mathbf{Exp}^{\mathsf{real}}_{\mathsf{uokms}}(\mathcal{A}, \ell)]$ and $p_1 = \Pr[1 \leftarrow \mathbf{Exp}^{\mathsf{ideal}}_{\mathsf{uokms}}(\mathcal{A}, \mathsf{SIM}, \ell)]$ 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. To reduce visual clutter we denote plaintext files as m instead of Obj and we omit idenfiers ObjId in ciphertexts.

We will first describe game $\mathbf{G}$, which reproduces the same distribution $\mathcal{A}$ sees in the real security game $\mathbf{Exp}^{\mathsf{real}}_{\mathsf{uokms}}$, 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 random group elements $g_1, ..., g_N$ in G, where N is the upper-bound on the number of Enc queries $\mathcal{A}$ makes. Then for every $i > 0$, $\mathbf{G}$ picks the following values: If $\mathcal{A}$ corrupts KmS in epoch i 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 KmS for two epochs in the row $\mathbf{G}$ also outputs $\Delta_i = k_{i-1}/k_i$.) If $\mathcal{A}$ corrupts StS in epoch i then $\mathbf{G}$ acts depending on which party $\mathcal{A}$ corrupted in epoch $i-1$: (case 1) If it was StS then $\mathbf{G}$ picks random $\Delta_i \leftarrow \mathbb{Z}_q$ and outputs $y_i \leftarrow {y_{i-1}}^{1/\Delta_i}$; (case 2) If it was KmS then $\mathbf{G}$ picks random $\Delta_{j+1,i} \leftarrow \mathbb{Z}_q$ and outputs $y_i \leftarrow {y_j}^{1/\Delta_{j+1,i}}$ where j was the last epoch when $\mathcal{A}$ corrupted StS *before* $\mathcal{A}$ corrupted KmS in epoch $i-1$. Let E_K be the set of epochs when $\mathcal{A}$ corrupts KmS and E_S the set of epochs when $\mathcal{A}$ corrupts StS. The above process defines value δ_i for each $i \in E_S$ 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 $(i-1) \in E_S$, or as $\delta_j \cdot \Delta_{j+1,i}$, if $j \in E_S$ and $\{j+1, ..., i-1\} \subseteq E_K$. Given these values, $\mathbf{G}$ services oracles Enc, Dec, and UEnc at epoch $i \in E_S$ (note that these calls are disallowed if $i \in E_K$) as follows:

- $\mathbf{G}$ replies to n-th call to $\mathsf{Enc}(m)$ with $c = (w, \mathsf{Enc}_{\mathsf{dek}}(m))$ where $w = (g_n)^{\delta_i}$ and $\mathsf{dek} = H(z)$ for $z = (g_n)^k$; (Note that $z = w^{k/\delta_i}$, 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 $\mathsf{UEnc}(t', c)$ for $c = (w, e)$ with (w', e) for $w' = w^{\delta_i/\delta_{t'}}$ if $t' \in E_S$, and $w' = (w^{\delta_i \cdot k_{t'}})^{1/k}$ if $t' \in E_K$.

The correctness of Enc and Dec responses follows because $k_i = k/\delta_i$, and as for UEnc, note that $k_i = k/\delta_i$, and the implicit update from epoch t' to epoch i is $\Delta_{t',i} = k_{t'}/k_i$, which together implies that $\Delta_{t',i} = (k_t \cdot \delta_i) \cdot (1/k)$. Thus game $\mathbf{G}$ reproduces the exact same view as security game $\mathbf{Exp}^{\mathsf{real}}_{\mathsf{uokms}}$.

Simulator SIM interacts with an ideal experiment $\mathbf{Exp}^{\mathsf{ideal}}_{\mathsf{uokms}}$ and executes the same algorithm as game $\mathbf{G}$ – including picking the initial key k and keys k_i if $i \in E_K$ and update-related values δ_i and $\Delta_{j,i}$ if $i \in E_S$ as described above (and defining corresponding δ_i's and k_i's). For handling oracles Enc and Dec, SIM resorts to the (stateful) simulator SIM_E assumed by the Receiver Non-Committing (RNC) property of the symmetric encryption Enc. First, when $\mathcal{A}$ sends t-th query m to oracle Enc in epoch $i \in E_S$, we put m at position t in list L, and SIM replies to $\mathcal{A}$ with $c = (w, e)$ for $w = (g_t)^{\delta_i}$ and e computed by SIM_E on input $(t, |m|)$. Second, when $\mathcal{A}$ sends u to Dec, SIM replies with $v = (u^{1/\delta_i})^k$ and 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 $\mathbf{Exp}^{\mathsf{ideal}}_{\mathsf{uokms}}$ to reveal message m at the t-th position in list L, sends (t, m) as the Reveal query to SIM_E, and given key dek as SIM_E's response, defines

$H(z) = \mathsf{dek}$. By the RNC property of Enc, pairs (dek, e) produced by SIM_E are computationally 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 $\mathbf{Exp}^{\mathsf{ideal}}_{\mathsf{uokms}}$) 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 (w, e) formed as $w = (g_{j_{Q+1}})^{\delta_i}$ s.t. $\mathcal{A}$ queries H on $z = (g_{j_{Q+1}})^k = w^{k_i}$. In other words, if there is ϵ difference between $Pr[1 \leftarrow \mathbf{Exp}^{\mathsf{real}}_{\mathsf{uokms}}(\mathcal{A}, \ell)]$ and $\Pr[1 \leftarrow \mathbf{Exp}^{\mathsf{ideal}}_{\mathsf{uokms}}(\mathcal{A}, \mathsf{SIM}, \ell)]$ 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 3. □

5 Threshold OKMS and UOKMS

The key management systems (particularly for storage applications) that motivate our work are often characterized by the large amounts of data they store as well as the value and long-lived nature of this data. The whole security of such an operation depends on the security of the KMS client keys, hence the importance of key rotation (as addressed by UOKMS) as a way to limit the bad effects of key exposure. Yet, the main priority is to prevent these keys from leaking in the first place. Fortunately, all the schemes presented in this paper lend themselves to *efficient distributed implementations* via the very efficient *Threshold OPRF* tdh-op [30] shown in Figure 6.

Key and server initialization. Key $k \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ is secret shared using Shamir's scheme with parameters n, t; server S_i, $i = 1, \ldots, n$, holds share k_i.

Threshold Oblivious Computation of $F_k(x)$.

- On input x, client $\mathcal{C}$ picks $r \leftarrow_{\mathrm{R}} \mathbb{Z}_q$ 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 $\mathcal{C}$, 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 $\mathcal{C}$ receives b_i from each server $S_i, i \in \mathcal{SE}$, $\mathcal{C}$ outputs as the result of $F_k(x)$ the value $H(x, (\prod_{i \in \mathcal{SE}} b_i)^{1/r})$.

Figure 6: Protocol tdh-op [30]: (n, t)-threshold computation of dh-op from Fig. 3

In our application, client keys k_c are shared among n KMS servers $S_1, \ldots, S_n$, so that the cooperation of $t+1$ of these is needed to compute the OPRF function with k_c as the key, while the compromise of t servers provides no information to the attacker on k_c. Moreover, the key k_c is never reconstructed or exists in one place, not even at generation (which is also performed distributively).

In addition, this scheme enjoys *proactive security* [44, 25], namely, the sharing among the n servers can be refreshed periodically so that the attacker needs to break into $t+1$ servers during the same time period to be able to compromise the key. Servers can be replaced and shares recovered, protecting secrecy and integrity/availability of the system as needed for long-lived keys.

The tdh-op function from Fig. 6 implements exactly the OPRF as defined in the OKMS scheme from Fig. 2. For the UOKMS scheme of Fig. 4, the only difference is in the input from the client (a random group element rather than a hashed value).

Note on efficiency. The dominant cost of computation in tdh-op is one exponentiation for each of the $t+1$ servers and two exponentiations for the client regardless of the values n and t. We note that tdh-op is described in a simplified form in Figure 6 where the set of reconstruction parties $\mathcal{SE}$ is assumed to be known by $\mathcal{C}$ 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 $\mathcal{C}$ 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 client.

5.1 Distributed Updates

While a threshold solution greatly increases the security of the KMS keys, one may still want to apply key rotation, particularly given the efficiency of updates in our UOKMS solution. In the threshold setting this means that at the beginning of a rotation epoch, the servers, that have a sharing of a key k_c, need to choose a new random client key $k_c{}'$ and generate the value $\Delta = k_c/k_c{}'$. However, Δ should only be disclosed to the client $\mathcal{C}$ and the storage server StS, calling for a distributed generation of Δ where no subset of t or less servers learn anything about this value.

We show a procedure that given (n,t) Shamir sharing of a key k generates (n,t) sharing of a new random key k' and of the update token $\Delta = k/k'$. It uses two standard tools from multi-party computation: (i) The joint generation of a Shamir sharing $\rho_1, \ldots, \rho_n$ of a uniformly random secret ρ over $\mathbb{Z}_q$, e.g., [46] or Fig. 7 of [23], and (ii) a Distributed Multiplication protocol which given the sharings of secret a and secret b generates a sharing of the product $a \cdot b$ without learning anything about either secret, e.g., [23].

The distributed update protocol assumes that n servers $S_1, \ldots, S_n$ have a sharing $(k_1, \ldots, k_n)$ of a key k. To produce a new key k' the servers jointly generate a sharing $\rho_1, \ldots, \rho_n$ of a random secret $\rho \in \mathbb{Z}_q$ and run distributed multiplication to generate shares $k'_1, \ldots, k'_n$ of the new key defined as $k' = \rho \cdot k$. Finally, each server S_i sends to $\mathcal{C}$ and/or StS its share ρ_i from which the recipient reconstructs ρ and sets $\Delta := \rho^{-1}$ $[= k'/k]$.

5.2 Verifiable Threshold (U)OKMS

As noted earlier, being able to verify the correctness of a data encryption key dek before encrypting an object is an important feature of the OKMS solution and a major advantage over traditional wrapping-based KM systems (Fig. 1). OKMS Verifiability requires checking the correct OPRF operation by the KmS for which Verifiable OPRFs [28] are available, assuming the client possesses the authentic public key g^{k_c} corresponding to KmS key k_c. As indicated in Section 2.2, verifiability in the case of UOKMS can be done directly via correct symmetric authenticated decryption, thus dispensing with the need to check the oblivious operation by KmS. In the threshold case, however, where multiple servers provide input for decryption, it is necessary to identify misbehaving servers. Thus, in the threshold case, verifiability is needed also for UOKMS.

Note that for the single-server dh-op scheme of Fig.3, verifiability can be added via a simple non-interactive zero-knowledge proof of equality of logarithms. For the threshold case, namely, tdh-op scheme of Fig.6, if we assume that the client possesses the public keys g^{k_i} corresponding to the shares k_i of key $k = k_c$, then zero-knowledge proofs can be used too for verification. However, this prevents the ability to have a "proxy" (e.g., any one of the n servers) that does the aggregation of the b_i values returned by the servers into the OPRF result. With ZK verification, it is the client itself that needs to do this aggregation. This loses the "client transparency" property of tdh-op that has the important practical advantage that the client (and its software) need not be aware of the implementation of the server, whether it is a single-server deployment or a multi-server one.

Next, we present an alternative verification procedure that is client transparent. The client only needs to have the certified public key g^k for key k (regardless of the number of servers). We first describe the scheme for the case of the single-server OPRF dh-op from Fig. 3 and later extend it to the threshold case. (This works directly for OKMS, the adaptation to UOKMS is immediate.) 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. It is easy to verify that 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 server 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
$$A^{r'} = B^{c'} v^{-dc'} \tag{1}$$
where $r' = r^{-1}, c' = c^{-1}$ (and $v = g^k$). It rejects if the equality does not hold, otherwise C sets the value of $(H'(x))^k$ to $A^{r'}$ which it is already computed for equation (1).

This procedure involves running dh-op on two different values and then verifying consistency via a single multi-exponentiation by the client. The additional computational cost with respect to the base dh-op is a single exponentiation for the server and two multi-exponentiations for the client, essentially doubling the work for the non-verified case.

We now adapt the scheme to the threshold OPRF tdh-op. 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 (1). If it holds, C sets $(H'(x))^k$ to $A^{r'}$, else it applies the check (1) to each pair A_i, B_i received by participating server S_i using $v_i = g^{k_i}$ instead of v.

The computational cost in the normal case, where the verification against $v = g^k$ succeeds, is the same as in 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. As said, the special feature of this procedure is that the client can interact with a proxy (or gateway) in a way that all operations by the client are identical to the single-server case. The proxy will send the values (a, b) generated by the client to the servers and will aggregate the responses A^i, B^i into a single response that can be verified with the public key g^k. Before sending to the client, the proxy can verify if the aggregation verifies correctly. If not, it needs to check the individual values sent by each server and discard the bad ones – all of this is done without any awareness by the client. Thus resulting in fully client-transparent solution.

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	-
Inverse Blind	16.95	16.95
Remove Blind	68.07	68.07
Verify Response	106.67	-
Total Time (μs)	402.86	212.92
Operations / Second	2,482	4,696

Figure 7: Client operation time and Op/s in OKMS

Interpolation Layer Performance (Single Thread)		
(t+1)-of-N	Time (μs)	Ops / Second
1-of-1	81.68	12,242
3-of-5	155.99	6,410
5-of-9	236.72	4,224
5-of-15	236.66	4,225
6-of-11	280.04	3,570

Figure 8: Interpolation layer performance for various threshold parameters

6 Implementation and Performance

We report on implementation and performance of the OKMS and UOKMS schemes from Section 2.1 (Fig. 2) and Section 2.2 (Fig. 4), 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 performing these operations concurrently across multiple CPU cores.

In these tests, all elliptic curve operations were based on NIST P-256. Field operations (for Shamir and blinding factors) were defined over the prime order of NIST P-256. Hashing to the curve (as required by the OPRF defined in Fig. 3) was performed using SHA-256 and the constant time *Simplified SWU* algorithm [12].

Client operations in the OKMS scheme are shown in Fig. 7 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.

When the (U)OKMS servers are deployed in a threshold architecture then some entity must perform polynomial interpolation "in the exponent". This can be implemented as a multi-exponentiation of the individual server's contributions together with the corresponding Lagrange coefficients. This interpolation operation could variously be performed by one of the servers, by the client, or by a dedicated intermediate entity.

(U)OKMS Server Operations (Single Thread)		
	Time (μs)	Ops / Second
Wrap	136.13	7,345
Unwrap	68.07	14,691

Figure 9: (U)OKMS Server performance for wrap and unwrap

UOKMS Operations (Single Thread)		
	Time (μs)	Operations / Second
Wrap	24.24	41,261
Unwrap	162.33	6,160
Update	68.07	14,691

Figure 10: Client operation time and Op/s in UOKMS

We observe that the multi-exponentiation time dominates the cost of the interpolation (computing the Lagrange coefficients is correspondingly cheap), and we find that the total cost depends on the threshold rather than the total number of servers. We report interpolation times in Fig. 8.

Server operations, measured for the single server and threshold variants of this (U)OKMS scheme (implemented with protocol tdh-pop) are shown in Fig. 9. This includes only 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 5.2).

Total time and operations per second is not significantly different for the threshold case, as each of the involved servers computes the same function in parallel.

Client performance in the UOKMS scheme (Sec. 2.2) is shown in Fig. 10. 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.

(U)OKMS Server. To evaluate performance and scalability we hosted our (U)OKMS server implementation on Amazon's Elastic Compute Cloud (EC2)[3] 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,

(U)OKMS Server Throughput (8 CPU cores)		
Scheme	KeepAlive	No KeepAlive
Static Page	65,018	6,462
OPRF (Unwrap)	32,094	6,349

Figure 11: (U)OKMS Server Requests/s on EC2 instance (LAN setting)

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 11 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.

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 OPRF/T-OPRF case. Thus our (U)OKMS implementation can handle a large number of clients with a single server. For comparison, Amazon's object storage service reported a peak load of 1.1 million requests per second [34]. If needed, the KMS implementation can be scaled with standard techniques, such as deploying a greater number of servers. With a few dozen servers in the KMS, a unique key could be supplied each time an object is written to or read from Amazon's service.

Hardware Security Modules. A best practice for securing master keys is to keep them within Hardware Security Modules (HSMs)[4] to prevent their export to less secure locations. Fortunately, the methods described in this paper are supported by existing commercial HSMs. Indeed, most HSMs support the PKCS#11 standard[43]. This specification defines an API method called `CKM_ECDH1_DERIVE` which takes an arbitrary point as an input and returns the x-coordinate of the point resulting from a scalar multiplication of the input point using an HSM-held private key as the scalar.

We tested three HSM implementations and found all supported the ECDH1 derive method. The returned x-coordinate is sufficient to perform an oblivious key derivarion, and verification, but verification (only needed in the OKMS setting and for threshold implementations of the OPRF) requires that both positive and negative solutions for the y-coordinate be checked By importing an elliptic curve private key computed as a Shamir share, existing HSMs can be used as part of a threshold implementation. Due to obliviousness, interpolating the result from a threshold of HSMs can be done external to the HSM without sacrificing confidentiality.

We note two potential limitations of using an HSM to hold the OPRF key. The first is that HSMs are often limited in the curves they support. While all of the HSMs we evaluated supported standard NIST curves, none supported Curve25519. The second limitation is performance. While high end commercial HSMs can achieve up to 22,000 scalar multiplications per second[49], this is roughly equivalent to what a single core can achieve in a multi-core server CPU.

While software implementations of symmetric key wrapping algorithms can be several orders of magnitude faster than asymmetric operations, we found HSMs often employ specialized hardware to accelerate the normally slower asymmetric operations. In some cases, HSMs[49] including those used by leading cloud providers[39], the number of supported ECC operations per second is comparable to that of supported symmetric encryption operations per second.

In conclusion, while in the traditional key-wrapping approach (Fig. 1) one can secure wrapping keys diligently e.g., in HSMs, the plain data encryption keys (dek) travel over much less secure TLS channels (sometimes ending or visible in multiple points outside the HSM boundary), and are potentially exposed to rogue administrators, accidental logging, etc. In contrast, these vulnerabilities are eliminated by the oblivious computation approach where as long as the OPRF key is secure, nothing can be learned about the data (other than by corrupting the client). Fortunately, securing these OPRF keys in HSMs is practical today as noted above, and while symmetric operations are less expensive than OPRF ones in general, HSMs with 20,000 EC op/sec can hardly be the system's bottleneck (of course, in large operations multiple HSMs will be used). Importantly, in UOKMS encrypting data does not necesitate of interaction with the KmS, further increasing performance. Additionally, the UOKMS approach offers much more efficient key rotation than traditional systems where rotation requires communication with the KMS for each key (dek or kek) to be updated. This slows down the rotation process, resulting in longer rotation periods and reduced security.

Acknowledgments

We thank Anja Lehmann for very helpful discussions related to security notions of Updatable Encryption schemes. Our implementation experience and reporting has benefited enormously from the work of Martin Schmatz, Navaneeth Rameshan, and Mark Seaborn. We thank the CCS reviewers who helped improving the presentation of the paper.

References

[1] 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.

[2] Amazon Web Services. Aws key management service cryptographic details, 2016. `https://d1.awsstatic.com/whitepapers/KMS-Cryptographic-Details.pdf`.

[3] Amazon Web Services. Aws elastic compute cloud, 2018. `https://aws.amazon.com/ec2/`.

[4] E. Barker and W. Barker. Recommendation for key management, part 2: Best practices for key management organizations (2nd draft). Technical report, National Institute of Standards and Technology, 2018.

[5] M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The one-more-RSA-inversion problems and the security of Chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185–215, June 2003.

[6] M. Bellare, C. Namprempre, D. Pointcheval, and M. Semanko. The One-More-RSA-Inversion problems and the security of chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185–215, 2003.

[7] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. In K. Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 127–144. Springer, Heidelberg, May / June 1998.

[8] D. Boneh, X. Boyen, and H. Shacham. Short group signatures. In M. Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 41–55. Springer, Heidelberg, Aug. 2004.

[9] 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.

[10] D. Boneh, K. Lewi, H. W. Montgomery, and A. Raghunathan. Key homomorphic prfs and their applications. *IACR Cryptology ePrint Archive*, 2015:220, 2015.

[11] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In P. Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 505–524. Springer, Heidelberg, Aug. 2011.

[12] 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`.

[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] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. In E. Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 255–271. Springer, Heidelberg, May 2003.

[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] A. Davidson, A. Deo, E. Lee, and K. Martin. Strong post-compromise secure proxy reencryption. In *Information Security and Privacy (ACISP) 2019*, 2019.

[18] D. Derler, S. Krenn, T. Lorünser, S. Ramacher, D. Slamanig, and C. Striecks. Revisiting proxy re-encryption: Forward secrecy, improved security, and applications. In M. Abdalla and R. Dahab, editors, *PKC 2018, Part I*, volume 10769 of *LNCS*, pages 219–250. Springer, Heidelberg, Mar. 2018.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Google Cloud. Google cloud key management service, 2018. `https://cloud.google.com/kms/`.

[25] 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.

[26] B. A. Huberman, M. K. Franklin, and T. Hogg. Enhancing privacy and trust in electronic communities. In *EC*, 1999.

[27] IBM. Ibm key protect, 2018. `https://console.bluemix.net/catalog/services/key-protect`.

[28] 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.

[29] 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.

[30] 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.

[31] S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu. TOPPSS: Cost-minimal password-protected secret sharing based on threshold OPRF. Cryptology ePrint Archive, Report 2017/363, 2017. `http://eprint.iacr.org/2017/363`.

[32] S. Jarecki, H. Krawczyk, and J. Resch. Updatable oblivious key management for storage systems. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*. ACM, 2019.

[33] 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.

[34] Jeff Barr. Amazon S3 – Two Trillion Objects, 1.1 Million Requests / Second, 2013. `https://aws.amazon.com/blogs/aws/amazon-s3-two-trillion-objects-11-million-requests-second/`.

[35] M. Klooß, A. Lehmann, and A. Rupp. (R)CCA secure updatable encryption with integrity protection. In *Eurocrypt 2109*, 2019.

[36] R. Lai, C. Egger, M. Reinert, S. Chow, M. Maffei, and D. Schröder. Simple password-hardened encryption services. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[37] 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.

[38] 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.

[39] Microsoft. How many cryptographic operations are supported per second with dedicated hsm?, 2019. `https://docs.microsoft.com/en-us/azure/dedicated-hsm/faq#performance-and-scale`.

[40] Microsoft Azure. Azure key vault, 2018. `https://docs.microsoft.com/en-us/azure/key-vault/key-vault-overview`.

[41] 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.

[42] 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.

[43] OASIS Open. PKCS #11 Cryptographic Token Interface Base Specification Version 2.40, 2015. `https://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html`.

[44] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In L. Logrippo, editor, *10th ACM PODC*, pages 51–59. ACM, Aug. 1991.

[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] Y. Polyakov, K. Rohloff, G. Sahu, and V. Vaikuntanathan. Fast proxy re-encryption for publish/subscribe systems. *ACM Transactions on Privacy and Security (TOPS)*, 20, 2017.

[48] 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.

[49] Thales. SafeNet Luna Network HSM, 2019. `https://safenet.gemalto.com/resources/data-protection/luna-sa-network-attached-hsm-product-brief/`.

A Proof of the OMDH-IO Assumption in the Generic Group Model

We sketch the steps for adapting the GGM proof of OMDH from [31] to the OMDH-IO case. As argued in [31], it suffices to show OMDH security for $N = Q + 1$, in which case the upper-bound on a probability that a GGM adversary solves the OMDH problem in a group of prime order q while making r group operations and Q queries to the exponentiation oracle $(\cdot)^k$ is $(Q(2Q + r)^2)/q$. In a GGM proof, see Theorem 6 in Appendix A in [31], every group element the adversary obtains is represented with a (random string assigned to) a polynomial in unknowns $(u_1, ..., u_N, k)$ for $u_i = \mathsf{DL}(g, g_i)$. Group multiplications or divisions correspond to, respectively, adding or subtracting such polynomials, and querying oracle $(\cdot)^k$ on a group element corresponds to multiplying the corresponding polynomial by k.

The proof argues that the only way the adversary can win is either if some two different polynomials it creates have equal values on random inputs $(u_1, ..., u_N, k)$, or that the group elements it outputs correspond to polynomials $k \cdot u_1, \ldots, k \cdot u_N$. The latter case is easily seen as impossible for an adversary which can has only $Q = N - 1$ accesses to the "multiply-a-polynomial-by-k" oracle $(\cdot)^k$, while the upper-bound on the probability of the first case comes from the fact that there are at most $2Q + r$ polynomials, each one has at most degree Q in k (and linear in variables $u_1, \ldots, u_N$), and the fact that a non-zero Q-degree polynomial can have at most Q roots, hence each pair of different polynomials can evaluate to the same value on random exponent $(u_1, ..., u_N, k)$ with probability at most Q/q. If the GGM adversary in addition makes t queries to the inverse-exponentiation oracle $(\cdot)^{1/k}$, each query multiplies the corresponding polynomial by k^{-1}, and the resulting polynomials, after multiplying all of them by k^t, can be thought of as polynomials of degree at most $Q + t$ instead of Q. Thus by the same argument, the upper-bound on the probability of GGM adversary to solve all $N = Q + 1$ challenges is bounded by $(Q + t)(2Q + t + r)^2/q$. Note that $r >> \max(Q, t)$ in typical applications, including our UOKMS scheme, hence this bound can be approximated as $(Q + t)r^2/q$.