An extended abstract [BDRW24a] of this article appears in the proceedings of ASIACRYPT 2024. The full version [BDRW24b] of this article is available in the IACR eprint archive.

Interval Key-Encapsulation Mechanism

Alexander Bienstock[1], Yevgeniy Dodis[2], Paul Rösler[3], and Daniel Wichs[4]

[1] J.P. Morgan AI Research and J.P. Morgan AlgoCRYPT CoE
alex.bienstock@jpmchase.com
[2] New York University dodis@cs.nyu.edu
[3] FAU Erlangen-Nürnberg paul.roesler@fau.de
[4] Northeastern University and NTT Research wichs@ccs.neu.edu

Abstract. *Forward-Secure* Key-Encapsulation Mechanism (FS-KEM; Canetti et al. Eurocrypt 2003) allows Alice to encapsulate a key k to Bob for some time t such that Bob can decapsulate it at any time $t' \leq t$. Crucially, a corruption of Bob's secret key after time t does not reveal k.

In this work, we generalize and extend this idea by also taking *Post-Compromise Security* (PCS) into account and call it *Interval Key-Encapsulation Mechanism* (IKEM). Thus, we do not only protect confidentiality of previous keys against future corruptions but *also* confidentiality of future keys against past corruptions. For this, Bob can regularly renew his secret key and inform others about the corresponding public key. IKEM enables Bob to decapsulate keys sent to him over an interval of time extending into the past, in case senders have not obtained his latest public key; forward security only needs to hold with respect to keys encapsulated before this interval. This basic IKEM variant can be instantiated based on standard KEM, which we prove to be optimal in terms of assumptions as well as ciphertext and key sizes.

We also extend this notion of IKEM for settings in which Bob decapsulates (much) later than Alice encapsulates (e.g., in high-latency or segmented networks): if a third user Charlie forwards Alice's ciphertext to Bob and, additionally, knows a recently renewed public key of Bob's, Charlie could re-encrypt the ciphertext for better PCS. We call this extended notion IKEMR. Our first IKEMR construction based on trapdoor permutations has (almost) constant sized ciphertexts in the number of re-encryptions; and our second IKEMR construction based on FS-PKE has constant sized public keys in the interval size.

Finally, to bypass our lower bound on the IKEM(R) secret key size, which must be linear in the interval size, we develop a new *Interval RAM* primitive with which Bob only stores a constant sized part of his secret key locally, while outsourcing the rest to a (possibly adversarial) server.

For all our constructions, we achieve security against active adversaries. For this, we obtain new insights on Replayable CCA security for KEM-type primitives, which might be of independent interest.

1 Introduction

Corruption of user secrets is an acknowledged threat in the cryptographic literature. Especially cryptographic protocols for secure long-term communication, such as secure messaging, implement measures to mitigate the effect of temporary user corruptions. The traditional goal *Forward Security* (FS) requires that prior communication remains secure even if user secrets are corrupted in the future. Modern communication protocols additionally fulfill *Post-Compromise Security* (PCS): users recover from earlier corruptions such that future communication will be secure again. Intuitively, this means that a corruption only reveals a (short) *interval* of secrets. To achieve these goals, protocols usually combine two techniques: (1) evolving secrets with one-way functions and then deleting the old secrets for FS, (2) randomly sampling fresh secrets and sharing the corresponding public values for PCS.

Session-Based Interval Security. The most prominent instantiation of this approach for two-party communication is the *Double Ratchet Algorithm* [PM16], implemented in Signal, WhatsApp, and many other messaging apps. A generalization based on tree-hierarchies for group communication is the *Messaging Layer Security* (MLS) standard [BBR+23].

Remarkably, both protocols and all their variants (e.g., [PR18a, PR18b, JS18, JMM19, RMS18, ACDT20, ACJM20, BDR20]) are session-based. This means that each new conversation is established independent of existing ones such that every user stores a separate collection of secrets for each session it participates in. Due to this, evolving and deleting old secrets as well as sampling fresh secrets is conducted separately per session, which induces a linear communication overhead in the number of sessions per user. For illustration, consider a user Bob who is concerned that all secrets locally stored on his device were corrupted. To recover from this corruption, Bob samples independent, fresh secrets and shares corresponding public values in all his sessions.

To reduce this overhead, Alwen et al. [AAB+21] investigate how to merge overlapping structures of session secrets in group communication protocols underlying MLS. Intuitively, they exploit that the tree-hierarchy in MLS can unify overlapping sets of members for multiple (independent) group sessions. Still, their saving is modest for most hierarchy structures, and does not generally result in sublinear storage in the number of sessions.

Interval Security with Single Key. In this work, we avoid any session separation and instead let each user have a consolidated secret key with a corresponding public key, which basically describes the concept of public-key encryption or, in our work, *Key-Encapsulation Mechanism* (KEM). Providing FS for KEM with a *static* public key—via FS-KEM—has already been solved by Canetti et al. [CHK03]. To also achieve PCS, the public key cannot remain static but must be updated. Thus, we will allow the recipient Bob to periodically update his public key. While this introduces some additional complexity in fetching Bob's latest key before sending him a message, we will see that this relaxation has many benefits, even beyond achieving PCS: for example, it results in noticeably more efficient schemes, by considerably simplifying achieving the FS aspect compared to the static public key model of [CHK03]. Additionally, our model will be sufficiently flexible to allow Bob to still decapsulate ciphertexts in many cases when the senders failed to obtain Bob's latest public key.

More concretely, our new model is the following. Bob starts by generating a key pair (sk_0, pk_0) and shares pk_0 with all users who want to talk to him. Whenever Bob thinks he was corrupted, he uses his current secret key sk_{i-1} to derive a new key pair (sk_i, pk_i) and shares the new pk_i again. Using sk_i, Bob can still decapsulate keys encapsulated with pk_j for $j \leq i$. However, if he thinks that all keys encapsulated with public keys $pk_j, j < i^*$ were decapsulated already or should not be decapsulatable any longer, Bob can shorten the secret key interval from $[0, i]$ to $[i^*, i]$. After this, Bob can only decapsulate ciphertexts encapsulated to a public key pk_j such that $i^* \leq j \leq i$. Conversely, security requires that a corruption of current secret key sk_i will not affect ciphertexts encapsulated to pk_j for $j < i^*$ or $j > i$. Since Bob can continuously renew his key pair to start new *epochs* and shorten the interval of decapsulatable old *epochs*, we call this primitive *Interval KEM* (IKEM).

KEM-Based IKEM. Our simple KEM-based IKEM construction almost naturally follows the above described abstract syntax: whenever Bob (re-)generates his IKEM key pair, he simply generates a fresh KEM key pair, shares the new KEM public key, and adds the new KEM secret key to his IKEM secret key. To shorten the decapsulation interval, Bob just removes old KEM secret keys from his IKEM secret key. The full details of this construction and a formal security analysis are in Section 3.2.

Secret Key Lower Bound. Notice, the extremely simple KEM-based IKEM from above has ciphertexts and public keys of constant size. However, the secret key grows linearly in the size of the current interval. One may wonder if this dependence is inherent. Unfortunately, as our first result we give the affirmative answer to this question: any IKEM secret key must be proportional to the size of the decapsulation interval.

One way to show this lower bound would be to prove that IKEM implies the simpler symmetric-key primitive called *Self Encrypted Queue* from Choi et al. [CDV21]. This primitive also involves a secret state that can be updated for PCS, however, it only allows the receiver to encrypt keys to *itself* for future use (hence why it is symmetric-key). Self Encrypted Queue also requires the newly updated states to be able to decrypt the keys encrypted to old states, and thus the state of any construction seemingly needs to grow proportionally to the number of epochs, just as with our IKEM construction. Indeed, [CDV21] shows that this is inherent for Self Encrypted Queues, by proving a lower bound showing that states need to be of size $f \cdot \lambda$, where f is the number of epochs.

Instead, we choose to prove a direct lower bound to show that IKEM secret keys need to grow with the size of the current interval, f, as we believe it more directly elucidates why this is in fact the case. Indeed, we use an encoding argument in Appendix B to accomplish this, with intuition as follows: The encoder's goal is to succinctly encode and send a random bit string s to the decoder, who then must obtain s. To this end, the encoder and decoder initially share public randomness (independent of s) consisting of an initial IKEM key pair, as well as two lists of f random bit strings each. To encode the ith bit of string s, the encoder selects the ith random bit string of one of the two lists—the first list *iff* $s_i = 0$. With each selected bit string, the encoder re-generates the current IKEM key pair. The final IKEM secret key sk^* is the code. The decoding algorithm also starts with the initial IKEM key pair. For every bit, it re-generates the current IKEM key pair twice: once with each next random bit string from the two lists. Using the two resulting public keys, it encapsulates individual keys and trial-decapsulates both resulting ciphertexts with the code sk^*. By correctness, decapsulation of the right ciphertext yields the matching encapsulated key. By security, decapsulation of the wrong ciphertext yields a random key (that is independent of the encapsulated key). This procedure is repeated with the right IKEM key pairs until string s is decoded entirely. The formal proof in Appendix B shows that the secret key size is linear in $f \cdot \lambda$, where f is size of the current decapsulation interval and λ is the security parameter.

Strengthening IKEM Security. While the lower bound on the secret key size of IKEM is unfortunate, in many settings the receiver can afford the extra storage, as this storage is local, and does not result in any increased network latency. Moreover, we will shortly describe a method to reduce the secret key storage by outsourcing. Yet, first we focus on extending the security of IKEM, by considering several motivating application scenarios.

First, in settings with high-latency or a segmented network topology, recovery by publishing a new public key for Bob may still be too slow: the new key may reach Bob's session partners only with a considerable delay. As an example, consider a decentralized gossiping or mesh network in which client devices exchange and forward traffic only with some contacts or with direct neighbors in their physical range (e.g., via bluetooth protocols). Based on this, a ciphertext from Alice to Bob is transmitted via multiple devices that span a delivery route in the network between them. Bienstock et al. [BRT23] observe that this topology allows contacts to cooperate with each other to strengthen the security of forwarded ciphertexts: if a user Charlie processes a ciphertext c from Alice to Bob after Charlie received the most recent public key for Bob, Charlie can re-encrypt c using the information contained in Bob's latest key. Thus, even if Alice sent c when Bob was corrupted, Charlie's re-encryption after Bob's key update protects c on the remaining delivery route from Charlie to Bob.

This form of contact cooperation can also strengthen security of encrypted cloud storage, where the cloud provider acts as Charlie. Every user Alice and Bob can upload encrypted data to Bob's online folder. Bob can regularly re-new his key material and share the latest public key with the provider. Without any further interaction, the provider can re-encrypt all files in Bob's folder such that they remain secure even if Bob's old secrets are ever corrupted. Note that this is the simplest, essential form of one of the motivating examples for Proxy Re-

Encryption (PRE) [BBS98].[5] We elaborate on the relation to PRE at the end of this section.

More generally, contact cooperation can strengthen any high-latency delivery or long-time storage during which ciphertexts are processed by intermediate honest parties. For example, Alice can encapsulate a ciphertext c_1 for Bob at time t_1, and leave it with an intermediary Charlie (e.g., a notary), instead of immediately delivering it to Bob. Charlie is then instructed to deliver c_1 to Bob some time t_2 in the future; e.g., if a certain condition is triggered. With traditional encapsulation, the key in c_1 would be compromised if Bob is corrupted any time between t_1 and t_2. With IKEM syntax, however, it might be possible for Charlie to update c_1 into a "more secure variant" c_1', using Bob's latest public key at epoch $t_2 > t_1$, and then only keep c_1' until it is released to Bob.

Ciphertext Re-Encapsulation. Motivated by this form of contact cooperation, we add re-encapsulation to IKEM and call it IKEMR. A ciphertext c_1 encapsulated to Bob at time t_1 and re-encapsulated in ciphertext c_1' at time $t_2 > t_1$ remains secure even if Bob was corrupted at any time $t^* < t_2$ as long as the adversary only ever sees c_1'. Furthermore, when Bob shortens the decapsulation interval of his secret key to exclude time t_1 for FS, future corruptions of Bob will not affect c_1' either. Thereby, re-encapsulations do not extend the lifetime of a ciphertext: if the epoch t_1 at which a ciphertext was *originally* created falls out of the decapsulation interval, it cannot be decapsulated anymore even if it was re-encapsulated at later epochs $t_2 > t_1$. We illustrate this security requirement with a simple example in Figure 1. Finally, a ciphertext can be re-encapsulated multiple times such that the first ciphertext version observed by the adversary determines the window of harmful and harmless corruptions, respectively. While this form of re-encapsulation ostensibly is closely related to Proxy Re-Encryption (PRE), we discuss the crucial differences at the end of this section.

Simple Extension Fails. Naively, one could add re-encapsulation to our KEM-based IKEM construction from above as follows: to re-encapsulate ciphertext c_1 from time t_1 with the most recent public key pk_2 from time $t_2 > t_1$, one simply uses the KEM to encapsulate a key k' in ciphertext c'_{KM} and then encrypts c_1 symmetrically with key k'. Thus, $c_1' = c'_{\mathrm{KM}} \| \mathrm{E}_{k'}(c_1)$ is the re-encapsulated ciphertext, where $(k', c'_{\mathrm{KM}}) = \mathrm{KM.enc}(pk_2)$. To shorten the decapsulation interval, Bob still just removes the KEM secret keys of all abandoned epochs from his IKEMR secret key.

Now consider the case that Alice creates two ciphertexts c_1 and c_2 to Bob, one at time t_1 with pk_1 and one at time $t_2 > t_1$ with pk_2. Additionally, Charlie re-encapsulates c_1 at time t_2 in ciphertext c_1'. When Bob shortens his decapsulation interval from $[0, t_2]$ to $[t_2, t_2]$, correctness requires that c_2 can still be decapsulated. Furthermore, security requires that k_1 in c_1' remains confidential if the adversary only sees c_1' (but not c_1), even if Bob is corrupted before time t_2 and/or after he shortened his decapsulation interval to $[t_2, t_2]$ (the reader can

[5] It is also a naturally useful variant of Updatable Encryption [BLMR13] for the public-key setting.

again refer to Figure 1 for an illustration of this security requirement). Yet, a corruption at time $t_1 < t_2$ for the above KEM-based IKEMR construction reveals KEM secret key sk_1; another corruption after the decapsulation interval was shortened to $[t_2, t_2]$ reveals KEM secret key sk_2. Thus, the adversary can remove the re-encapsulation layer of c'_1 using sk_2 and then decapsulate the original ciphertext c_1 using sk_1. Hence, this simple extension of our KEM-based IKEM construction is insecure.

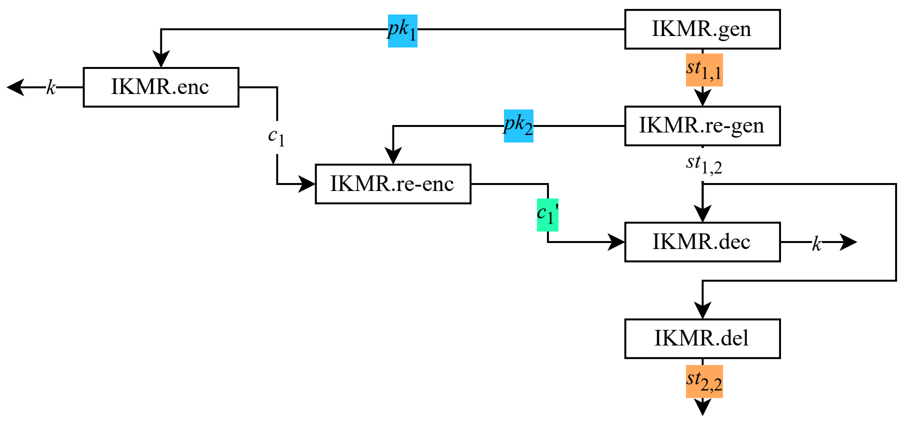

Fig. 1. Execution example for IKEMR: if the adversary only sees pk_1, pk_2, and c'_1, and corrupts $st_{1,1}$, and $st_{2,2}$, key k is required to remain secure.

KEM-Based IKEMR. The core problem of the above IKEMR construction is that it processes encapsulations and *re*-encapsulations at time t_2 identically. However, FS requires that Bob can shorten his decapsulation interval such that only c_2, originally created at time t_2, can be decapsulated but not the re-encapsulation c'_1 from time t_2 of earlier ciphertext c_1, originally from time t_1.

To solve this problem, Bob, instead, generates multiple *extra* KEM key pairs at every IKEMR key re-generation: one KEM key pair for each epoch in his current decapsulation interval of length f. The resulting re-generated IKEMR public key $pk_t = (pk_{t,1}, \ldots, pk_{t,f})$ for time t consists of f fresh KEM public keys. (Re-)Encapsulation with IKEMR public key pk_t at time t then depends on the initial encapsulation time of a ciphertext: for ciphertext c initially created at time $t^* \leq t$ ($t^* = t$ for an initial encapsulation), c is (re-)encapsulated using KEM public key $pk_{t,t-t^*+1}$. Whenever Bob shortens his decapsulation interval to $[t_1, t_2]$, he first removes all secret keys generated before time t_1. Then, for each epoch t' with $t_1 \leq t' \leq t_2$, he removes every secret key $sk_{t',\tau}$ for which $\tau > t' - t_1 + 1$. Thus, intuitively, every epoch's IKEMR public key contains extra KEM public keys that are exclusively used for *re*-encapsulations of ciphertexts initially created in prior epochs. As soon as such prior epochs are removed from the decapsulation interval, all corresponding extra KEM secret keys are removed from the IKEMR secret key.

By deriving the extra KEM secret keys at each IKEMR re-generation iteratively via a chain of Pseudo-Randomness Generators (PRGs), Bob only ever needs to store one KEM secret key and one PRG seed for each epoch in his current decapsulation interval. To shorten the decapsulation interval, Bob moves the PRG-chain of each epoch in the interval forward, thereby derives new KEM secret keys and deletes past KEM secret keys. Thus, the IKEMR secret key size is optimal based on our lower bound.

IKEMR WITH SMALL PUBLIC KEYS. The advantage of this secure IKEMR construction is its mild underlying standard assumption: an ordinary KEM. Nevertheless, the disadvantages are: ciphertexts grow linearly in the number of re-encapsulations and public keys grow linearly in the decapsulation interval length. Using FS-PKE instead of ordinary KEM, we can reduce the size of IKEMR public keys. The intuition is that all *extra* KEM key pairs generated for one IKEMR key re-generation are consolidated in a single FS-PKE key pair. This shrinks the IKEMR public key to constant size but slightly increases the secret key to size $f \log f$ (instead of f), where f is the current length of the decapsulation interval. At (re-)encapsulation, instead of choosing the right *extra* KEM public key from the current IKEMR public key, Alice indicates the ciphertext's initial creation time when encrypting to the corresponding epoch of the single FS-PKE public key. When shortening the decapsulation interval, Bob simply updates all FS-PKE secret keys that remain in his IKEMR secret key such that epochs outside the shortened interval cannot be (re-)decapsulated. The full details of this construction are in Section 6.

IKEMR WITH SMALL CIPHERTEXTS. Using Trapdoor Permutations (TDPs), we build an IKEMR construction with (almost) constant sized ciphertexts. For this, we begin with our KEM-based IKEMR construction from above that has linear sized public keys in the decapsulation interval length. Then, instead of using KEMs, we employ a family of TDPs with a common domain, where each KEM public key is replaced with the public key of a TDP from that family and each KEM secret key is replaced with the corresponding trapdoor. To encapsulate a key, this construction samples a random element from the TDP family's domain and evaluates the current epoch's TDP on it, which yields the ciphertext. The actual encapsulated key is derived by applying a randomness extractor to the random input element. To re-encapsulate a ciphertext, another TDP from the family is applied to this ciphertext. Since all permutations from the TDP family share the same domain, the (cryptographic part of the) ciphertext has constant size. Yet, for decapsulation, the receiver Bob needs to know which trapdoors he should use. Thus, at each re-encapsulation, the index of the current epoch is attached to the ciphertext, which increases the ciphertext linearly in the number of re-encapsulations—however, note that each attached index has only logarithmic size in the security parameter.

Reducing Ciphertext Size. To further reduce the IKEMR ciphertext size for settings with frequent re-encapsulations, we add more TDPs to the IKEMR public key in order to avoid attaching epoch indices to re-encapsulated ciphertexts. Note that in our KEM-based IKEMR construction, each IKEMR public key consists of one f-sized batch of KEM public keys. For our above TDP-based IKEMR construction, these KEM public keys are replaced with TDP public keys. Now, for this final construction, each IKEMR public key consists of Δ batches of f TDP public keys. These additional batches are used when Charlie, at time t_2, wants to re-encapsulate a ciphertext originally created at time t_1, where $d = t_2 - t_1 \leq \Delta$. In this case, instead of directly re-encapsulating with the newest TDP public key and informing Bob about the re-encapsulation 'jump' from time t_1 to time t_2, Charlie continuously applies d TDP evaluations, one after another: from t_1 to $t_1 + 1$ to $t_1 + 2$, and so on until t_2. Thus, when Bob decapsulates, he can simply iterate over a continuous chain of trapdoor inversions in reverse. Only for larger re-encapsulation jumps of size $d > \Delta$, Charlie attaches epoch indexes to the ciphertext. Intuitively, if f stays roughly the same throughout the execution of the IKEMR, then public keys will be of size $O(f \cdot \Delta)$ and ciphertexts will be of size $O(\lambda + \log(\lambda) \cdot f/\Delta)$, regardless of the number of re-encapsulations; in particular, if $\Delta = f$, then ciphertexts will always be of size $O(\lambda)$, even after several re-encapsulations.

External Key Storage. Recall from above that the secret key of any IKEM must have size $f \cdot \lambda$, where f is the size of the current decapsulation interval. For long intervals or even intervals of dynamic size, this lower bound may induce an impractical storage overhead. Thus, we propose to split the secret key into two components—one small component and one larger component, of which only the former needs to be securely stored. The latter component, on the other hand, can be stored anywhere, and can even be publicly accessible. We furthermore desire a generic interface such that reads and writes to the original (virtual) secret key can still be efficiently performed. Indeed, such operations should only use the small securely-stored component, as well as small downloads from and uploads to the larger public component. Most importantly, we still want the security properties of IKEM to hold if the small securely-stored component is corrupted from time to time.

The notion of securely outsourcing a database with read- and write-access with *only FS* is very well-studied (e.g., [BDT22, BDY21, BL96], and references therein). Indeed, a construction for n-entry databases satisfying this notion is known such that the secret state is size $O(1)$ and the read/write overhead is $O(\log n)$. However, the security models from these prior works do not require secrecy of future writes if at any time the secret storage is corrupted. That is, to the best of our knowledge, no notion of securely outsourcing a database with *PCS* is known.

In Section 7, we introduce the notion of securely outsourcing a database with read- and write-access with *both* FS and PCS, and call it Interval RAM (IRAM). We then show that the construction mentioned above surprisingly satisfies this stronger notion of security. IRAM can be combined with our IKEM constructions to reduce the size of local secret storage, while maintaining security.

IKEMR vs. Proxy Re-Encryption. A primitive related to IKEMR is Proxy Re-Encryption (PRE). PRE extends standard public-key encryption with a key-based re-encryption mechanism: every secret key sk_A can compute a re-encryption key $rk_{A,B}$ for re-encryption to another public key pk_B. Using $rk_{A,B}$, ciphertext c_A, previously (re-)encrypted to public key pk_A, can be re-encrypted to c_B such that c_B behaves as if it was encrypted to pk_B without changing the payload.

PRE schemes can be bidirectional or unidirectional. With bidirectional PRE, a re-encryption key $rk_{A,B}$ for epochs A and B can be used to re-encrypt from A to B *and* vice versa. Thus, given sk_A and $rk_{A,B}$, there are naturally no security properties regarding ciphertexts encrypted to pk_B. For unidirectional PRE, re-encryption key $rk_{A,B}$ can *only* be used to re-encrypt from A to B, and given sk_A and $rk_{A,B}$, ciphertexts encrypted to pk_B are still secure (see, e.g., [DDLM19, FKKP19, MPW23]).

Bidirectional PRE is clearly insufficient for secure IKEMR. Yet, IKEMR intuitively seems weaker than unidirectional PRE for three reasons (a formal lower bound is a harder task, which we deem out of scope): (1) In unidirectional PRE, re-encryption keys are only derived from the old secret key and the new *public* key; in IKEMR, the public keys used for re-encapsulation are derived from *both* old secret keys and the new secret key. (2) In IKEMR, ciphertexts can only be re-encapsulated to newer public keys; in unidirectional PRE, re-encryption keys can be derived for arbitrary public keys, which may even lead to full re-encryption circles. (3) Unidirectional PRE offers additional security guarantees if the re-encryption key remains secret—IKEMR public keys are, by definition, always public.

Indeed, we achieve IKEMR with constant-size ciphertexts from TDPs, while unidirectional PRE with constant-size ciphertexts can only be achieved from (expensive) FHE or iO currently (see, e.g, [MPW23, p. 11]). Furthermore, even unidirectional PRE seems unsuitable for building IKEMR with FS directly: an unlimited chain of PRE re-encryption keys can be used to shift an old ciphertext to a much newer, corrupted secret key, which undermines FS.

Further Related Primitives. Like IKEM, primitives such as Updatable PKE (UPKE) [JMM19, DKW21, HPS23] and Key-Updatable KEM (KU-KEM) [PR18b, BRV20, RSS23] continuously update both parts of the key pair. Yet, these primitives only achieve FS and they are designed to work in a session-based fashion between a fixed tuple of Alice and Bob. Thereby, since the public keys in UPKE and KU-KEM are continuously updated by *Alice*, this would require synchronization when multiple *Alices* want to talk to the same Bob. IKEM overcomes this issue by letting all Alices use the newest public key they are aware of.

An orthogonal security feature that we do not cover in this work is in-epoch FS: the granularity of FS in IKEM is relatively coarse as only shrinking the interval by deleting entire epochs yields FS with respect to these deleted, old epochs. Using FS-KEM techniques—based on building blocks like identity-based encryption—, each epoch could have internal sub-steps for which FS can be achieved by updating Bob's secret key without invalidating the corresponding epoch entirely. We refrain from studying this aspect as it seems to be a simple, straight forward extension.

Active Security. For all our constructions, we prove security against active adversaries. Before we can do so, we develop a suitable security definition that models Chosen Ciphertext Attacks (CCA) by giving adversaries access to a decapsulation oracle. Upon a queried ciphertext c, this oracle honestly decapsulates the symmetric key k encapsulated in c using Bob's current secret key, unless c was posed as a *challenge*. Clearly, decapsulating the challenge ciphertext c^* via this oracle trivializes the adversary's ability to win the security experiment.

Due to re-encapsulations, muting the decapsulation oracle upon challenge ciphertexts is, however, non-trivial: the adversary can re-encapsulate challenge c^* using Bob's newer public keys, which yields c^{**}. Thus, the decapsulation oracle has to reject c^*, all of its re-encapsulations c^{**}, and so on. The literature on PRE developed several approaches for identifying such re-encapsulated challenges c^{**} of which we consider only one suitable: employing a Replayable CCA (RCCA) [CKN03] definition style. However, the RCCA definition style is not directly applicable to KEM-type primitives. For this reason, we develop a suitable variant of RCCA security that is implied by CCA security for KEM-type primitives and can be used to build PKE that is RCCA-secure via standard hybrid encryption. We also show that our RCCA notion for IKEMR can be used to build the natural extension of this primitive that captures encryption, which we call *Interval Public Key Encryption with Re-encryptions* (IPKER). We elaborate on our definitional choices in Appendix C. To add RCCA security to our constructions, we use standard techniques from the literature, which add minimal overhead.

We also extend our IRAM security notion to withstand active adversaries. Here, the receiver (with small local storage) must be able to detect if the adversary provides them with incorrect parts of the (larger) public storage for a given operation. To achieve this notion, we combine our original construction with techniques from the Memory Checking literature ([BEG$^+$91, DNRV09, BDY21] and the references therein), while adding minimal overhead.

Contributions. In summary, we develop a natural notion of KEM that offers FS and PCS guarantees against active adversaries. For this, we prove a lower bound in Section 3 to show that the most basic construction is optimal. We extend the initial notion of IKEM by adding re-encapsulation, which we call IKEMR (see Section 4). Realizing IKEMR turns out to be more complicated: Our first construction in Section 5 uses Trapdoor Permutations to keep ciphertexts small. Our second construction in Section 6 uses FS-PKE to reduce the size of public keys. Finally, in Section 7, we introduce Interval RAM with which secret keys can be split into a locally stored part of constant size and a larger part that can be outsourced to an insecure (and actively adversarial) external storage.

2 Preliminaries

In Appendix A, we provide some definitions for basic primitives used in our IKEM constructions, as well as some standard definitions and lemmas from information theory. Below, we present some notation we will use throughout this work and then three important primitives which we will use in our constructions.

Notation We use $x \leftarrow y$ for assigning value y to variable x. We use $x \leftarrow_\$ \mathcal{X}$ to denote sampling x randomly from distribution $\mathcal{X}$. Consider some algorithm A. If A is deterministic, we use $y \leftarrow A(x)$ to denote assigning to y the output of $A(x)$. If A is randomized, we use $y \leftarrow_\$ A(x)$ to denote assigning to y the output of a random run of $A(x)$. Sometimes, we may explicitly specify the random coins r that a randomized algorithm A uses; in this case, we use $y \leftarrow A(x; r)$ to denote assigning to y the output of a run of $A(x)$ using coins r.

Family of Lossy Trapdoor Permutations with a Common Domain. We now define families of lossy trapdoor permutations (TDPs), in which all permutations in the family share a common domain $\mathcal{X}$ [AKPS19]. Lossy TDPs can be instantiated in *injective* or *lossy* mode—in injective mode, every input $x \in \mathcal{X}$ permuted to $y \leftarrow \mathrm{P.eval}(pk, x)$ can be inverted back to $x \leftarrow \mathrm{P.inv}(sk, y)$; in lossy mode, the image of $\mathrm{P.eval}(pk, \cdot)$ is much smaller than $\mathcal{X}$ (and therefore, finding x from $\mathrm{P.eval}(pk, x)$ is statistically-hard). Furthermore, for any adversary with just the public key pk, it is hard to distinguish whether pk was sampled in injective or lossy mode.

Syntax A family of *Lossy Trapdoor Permutations with Common Domain* (TDP) scheme P is a tuple of algorithms $\mathrm{P} = (\mathrm{P.gen}, \mathrm{P.eval}, \mathrm{P.inv})$ with the following syntax:

- $\mathrm{P.gen}(1^\lambda, b) \rightarrow_\$ (sk, pk)$ generates a key-pair. Input $b \in \{0, 1\}$ specifies whether the generated instance is *injective* ($b = 1$) or *lossy* ($b = 0$).
- $\mathrm{P.eval}(pk, x) \rightarrow y$ takes in a public key pk and input x and permutes x to y.
- $\mathrm{P.inv}(sk, y) \rightarrow x$ takes in a secret key sk and permuted output y, and inverts it to x (looking ahead, when in lossy mode, there are no properties required).

Correctness A family of lossy TDPs is *correct* if for any key pair sampled in *injective mode*, inputs x permuted to y can always be inverted to x.

Definition 1. *Scheme* P *is* correct *if for all* $(sk, pk) \leftarrow_\$ \mathrm{P.gen}(1^\lambda, 1)$ *and* $x \in \mathcal{X}$, $x = \mathrm{P.inv}(sk, \mathrm{P.eval}(pk, x))$.

Security For security of families of lossy TDPs, we require two properties. The first property is that when in lossy mode, the size of the image of $\mathrm{P.eval}(pk, \cdot)$ is much smaller than the domain $\mathcal{X}$.

Definition 2. *Scheme* P *is* L-lossy *if for all* $(sk, pk) \leftarrow_\$ \mathrm{P.gen}(1^\lambda, 0)$, $|\mathrm{P.eval}(pk, \cdot)| \leq |\mathcal{X}|/L$, *where* $|\mathrm{P.eval}(pk, \cdot)|$ *is the number of unique outputs across* $x \in \mathcal{X}$.

The second property is that an adversary that is given some sampled pk should not be able to tell if it was sampled in injective or lossy mode.

Definition 3. *Scheme* P *is* $(T, \varepsilon_{\mathrm{P}})$-secure *if for all adversaries $\mathcal{A}$ running in time T:* $\Pr[b \leftarrow_{\$} \mathcal{A}(pk) : b \leftarrow_{\$} \{0,1\}; (sk, pk) \leftarrow_{\$} \mathrm{P.gen}(1^{\lambda}, b)] \leq 1/2 + \varepsilon_{\mathrm{P}}$.

Auerbach et al. show how to construct families of Lossy TDPs with a common domain $\mathcal{X} = \{0,1\}^n$ from many assumptions [AKPS19]. Of note for our purposes, they construct such a Lossy TDP family with lossiness $L = 2^{n/4}$ from the Phi-Hiding Assumption.

All-But-One Trapdoor Functions. We now define families of all-but-one (ABO) trapdoor functions, which are a generalization of lossy trapdoor functions. In an ABO family, each function has several *branches*. All of the branches are injective, except for one branch that is lossy. Moreover, an adversary with the public key pk of the ABO cannot tell which of the branches is lossy.

Syntax A family of *all-but-one trapdoor functions* (ABO) scheme ABO is a tuple of algorithms ABO = (ABO.gen, ABO.eval, ABO.inv) with the following syntax:

- $\mathrm{ABO.gen}(1^{\lambda}, b) \rightarrow_{\$} (sk, pk)$ generates a key-pair. Input $b \in \{0,1\}^v$, for $v \in \mathsf{poly}(\lambda)$ specifies the branch that is *lossy*.
- $\mathrm{ABO.eval}(pk, b, x) \rightarrow y$ takes in a public key pk, branch b, and input x and outputs y.
- $\mathrm{ABO.inv}(sk, b, y) \rightarrow x$ takes in a secret key sk, branch b, and output y, and inverts it to x (for lossy branch b, there are no properties required).

Correctness A family of ABOs is *correct* if for any key pair, inputs x mapped to y for any *injective* branch can always be inverted to x (under the same branch).

Definition 4. *Scheme* ABO *is* correct *if for all* $b \neq b' \in \{0,1\}^v$*,* $(sk, pk) \leftarrow_{\$} \mathrm{ABO.gen}(1^{\lambda}, b)$*, and* $x \in \mathcal{X}$*,* $x = \mathrm{ABO.inv}(sk, b', \mathrm{ABO.eval}(pk, b', x))$*.*

Security For security of families of ABOs, we require two properties. The first property is that for the lossy branch b of the function, the size of the image of $\mathrm{ABO.eval}(pk, b, \cdot)$ is much smaller than the domain $\mathcal{X}$.

Definition 5. *Scheme* ABO *is* L-lossy *if for all* $b \in \{0,1\}^v$ *and* $(sk, pk) \leftarrow_{\$} \mathrm{ABO.gen}(1^{\lambda}, b)$*,* $|\mathrm{ABO.eval}(pk, b, \cdot)| \leq |\mathcal{X}|/L$*, where* $|\mathrm{ABO.eval}(pk, b, \cdot)|$ *is the number of different outputs across all inputs* $x \in \mathcal{X}$*.*

The second property is that an adversary that is given some sampled pk should not be able to tell which branch is lossy.

Definition 6. *Scheme* ABO *is* $(T, \varepsilon_{\mathrm{ABO}})$-secure *if for any* $b_0, b_1 \in \{0,1\}^v$*, for all adversaries $\mathcal{A}$ running in time T:* $\Pr[\delta \leftarrow_{\$} \mathcal{A}(pk) : \delta \leftarrow_{\$} \{0,1\}; (sk, pk) \leftarrow_{\$} \mathrm{ABO.gen}(1^{\lambda}, b_{\delta})] \leq 1/2 + \varepsilon_{\mathrm{ABO}}$.

Peikert and Waters, show how to construct a family of ABOs with domain $\mathcal{X} = \{0,1\}^n$ and lossiness $L = 2^n/\lambda$ from the DDH assumption [PW08].

Forward-Secure Public-Key Encryption. We briefly define FS-PKE [CHK03], which is close to our definition of f-Bounded Forward-Secure Lossy TDPs with Common Domain in Section 5.

Syntax Forward-Secure Public-Key Encryption (FSE) is a tuple of algorithms $\mathrm{FSE} = (\mathrm{FSE.gen}, \mathrm{FSE.up}, \mathrm{FSE.enc}, \mathrm{FSE.dec})$ with the following syntax:

– $\mathrm{FSE.gen}(1^\lambda, f) \rightarrow_\$ (SK_0, PK)$ on input f that specifies the maximal number of update-epochs, outputs initial secret key SK_0 and public key PK.
– $\mathrm{FSE.up}(SK_t) \rightarrow SK_{t+1}$ updates input secret key SK_t to SK_{t+1}.
– $\mathrm{FSE.enc}(PK, t', m) \rightarrow_\$ c$ on input PK and epoch t', encrypts m for epoch t' in c.
– $\mathrm{FSE.dec}(SK_t, t', c) \rightarrow m$ on input SK_t and t', decrypts c for epoch t' to output m.

Note that t' is an explicit input of FSE.dec(). For our usage of FSE.dec() within our IKEMR construction, we will be able to extract the proper t' from the rest of the ciphertext.

Correctness Correctness requires that, for $t_0 \leq t_1$, the receiver with SK_{t_0} can decrypt $c \leftarrow \mathrm{FSE.enc}(PK, t_1, m)$ to m.

Definition 7. *Scheme* FSE *is* correct *if for all* $(SK_0, PK) \leftarrow_\$ \mathrm{FSE.gen}(1^\lambda, f)$, *for every* $t \in [f-1], SK_t \leftarrow \mathrm{FSE.up}(SK_{t-1})$, *all* $m \in \mathcal{M}$ *and any* $0 \leq t_0 \leq t_1 \leq f-1$, $m = \mathrm{FSE.dec}(SK_{t_0}, t_1, \mathrm{FSE.enc}(PK, t_1, m))$.

Security For any adversary with only the public key PK and *any* secret key $SK_{t'}$ for $t' > t^*$, we require that it is hard to tell which of two same-length messages was encrypted to epoch t^*:

Definition 8. *Scheme* FSE *is* $(T, \varepsilon_{\mathrm{FSE}})$-secure *if for all adversaries* $\mathcal{A}$ *running in time* T*:*

$$\begin{aligned}\Pr[b \leftarrow_\$ \mathcal{A}^{\mathbf{Dec}^{\neq t^*}_{\neq c^*}}(SK_{t'}) : b \leftarrow_\$ \{0,1\}; f \leftarrow_\$ \mathcal{A}(); (SK_0, PK) \leftarrow_\$ \mathrm{FSE.gen}(1^\lambda, f);\\ (0 \leq t^* \leq f-1, m_0, m_1) \leftarrow_\$ \mathcal{A}^{\mathbf{Dec}}(PK); |m_0| = |m_1|;\\ c^* \leftarrow_\$ \mathrm{FSE.enc}(PK, t^*, m_b); (t^* < t' \leq f) \leftarrow_\$ \mathcal{A}^{\mathbf{Dec}^{\neq t^*}_{\neq c^*}}(c^*);\\ \textit{for } t \in [f-1], SK_t \leftarrow \mathrm{FSE.up}(SK_{t-1})] \leq 1/2 + \varepsilon_{\mathrm{FSE}},\end{aligned}$$

where decryption oracle $\mathbf{Dec}^{\neq t^*}_{\neq c^*}$ *on input* (t, t°, c) *outputs* $\mathrm{FSE.dec}(SK_t, t^\circ, c)$, *unless* $t \leq t^\circ = t^*$ *and* $c = c^*$.

3 Basic Interval Key-Encapsulation Mechanism

In this section, we introduce our basic Interval Key-Encapsulation Mechanism (IKEM) notion without re-encryptions. We provide a security definition with a decryption oracle and CCA security, but also, by simply removing the decryption oracle, provide a CPA security notion. Then, we show that the size of the secret state of any IKEM protocol must be proportional to the interval size (following from a lower bound of [CDV21] for a different, simpler, symmetric-key primitive). Importantly, this lower bound also holds for any IKEMR scheme (because any IKEMR without the re-encapsulation algorithm trivially implies an IKEM). Finally, we present a simple and efficient IKEM protocol based on a generic KEM, where CPA-(resp. CCA-)security is achieved if the KEM is CPA-(resp. CCA-)secure. We begin by defining the IKEM primitive.

3.1 IKEM Definition

The syntax of IKEM is the same as IKEMR, except without the re-encapsulation algorithm IKMR.re-enc().

Correctness An IKEM scheme is *correct* if secret state st_{t_0,t_1}, can decapsulate correctly any ciphertext created in any epoch $t \in [t_0, t_1]$. More formally:

Definition 9. *Given $T \in \mathbb{N}$, and dictionary D s.t. $D[i] = (t^i, \ell_i)$ for $i \in [m]$ containing items in $[T] \times [T]$ s.t. $0 < t^1 \leq \cdots \leq t^m \leq T$ and $\sum_{j=1}^{i} \ell_j \leq t^i$ for every $i \in [m]$: let $t_0 \leftarrow 0$; $(st_{0,0}, pk_0) \leftarrow_\$ \mathrm{IKM.gen}(1^\lambda)$; and for all $t_1 \in [T]$, $(st_{t_0,t_1}, pk_{t_1}) \leftarrow_\$ \mathrm{IKM.re\text{-}gen}(st_{t_0,t_1-1})$ and for every $i \in [m]$ s.t. for $(t^i, \ell_i) \leftarrow D[i]$, $t^i = t_1$, $st_{t_0+\ell_i,t_1} \leftarrow \mathrm{IKM.del}(st_{t_0,t_1}, \ell_i)$. Scheme* IKM *is* correct *if for every such T and D, as well as $t \in [t_0, t_1]$ and $(k, c) \leftarrow_\$ \mathrm{IKM.enc}(pk_t)$, $k = \mathrm{IKM.dec}(st_{t_0,t_1}, c)$.*

Security Now we define security for an IKEM scheme. At a high level, any key encapsulated to the public key pk_t for epoch t should be indistinguishable from random, even if the adversary gets multiple states st_{t_0,t_1} such that $t \notin [t_0, t_1]$. Additionally, for CCA security, the adversary can see decryptions of any ciphertext besides the challenge ciphertext.

More formally, we define the security game for IKEM in Figure 2. For the lower and upper bounds for IKEM that we will provide later in this section, we will define a one-way notion $\mathrm{OW}_{\mathrm{IKM}}$ and an indistinguishability notion $\mathrm{IND\text{-}X}_{\mathrm{IKM}}$, respectively (in the same figure). The red text in Figure 2 only applies to the indistinguishability notion. The security game starts by sampling a key pair via IKM.gen and returning the public key to the adversary. It also initializes a number of variables, including $t_X \leftarrow -\infty$, which will be used to store the latest epoch in which the secret state has been exposed. Oracle **Re-Gen**() executes IKM.re-gen on the current state and returns the new public key. Oracle **Del**(ℓ) executes IKM.del on the current state and adversarially-chosen ℓ, only if the updated interval would still be valid; i.e., $t_0 + \ell \leq t_1$. Oracle **Chall** first checks that the newest state has not been exposed; i.e., $t_1 \neq t_X$. If so, it executes IKM.enc on the newest public key pk_{t_1} to get key k_0 and ciphertext c^*. Then, for the indistinguishability game, the oracle samples key k_1 and bit b, then returns (k_b, c^*). For the one-way game, the oracle just returns c^*. In both games,

Initialization: Set $t_X, t^* \leftarrow -\infty$ and $t_0, t_1 \leftarrow 0$. Then compute $(st_{t_0,t_1}, pk_{t_1}) \leftarrow_\$$ IKM.gen(1^λ) and output pk_{t_1}.

Re-Gen():

```
1. increment $t_1 \leftarrow t_1 + 1$
2. regenerate $(st_{t_0,t_1}, pk_{t_1}) \leftarrow_\$$
     IKM.re-gen$(st_{t_0,t_1-1})$
3. return $pk_{t_1}$
```

Dec(c):

```
1. return $\perp$ if $c = c^*$
2. return IKM.dec$(st_{t_0,t_1}, c)$
```

Del(ℓ):

```
1. return $\perp$ if $t_0 + \ell > t_1$
2. increment $t_0 \leftarrow t_0 + \ell$
3. compute $st_{t_0,t_1} \leftarrow$
     IKM.del$(st_{t_0-\ell,t_1}, \ell)$
```

Chall():

```
1. return $\perp$ if $t_1 = t_X$
2. set $t^* \leftarrow t_1$
3. encapsulate $(k_0, c^*) \leftarrow_\$$ IKM.enc$(pk_{t_1})$
4. sample random $k_1 \leftarrow_\$ \mathcal{K}$
5. flip random coin $b \leftarrow_\$ \{0, 1\}$
6. disable Chall() and return $(k_b, c^*)$
```

Expose():

```
1. return $\perp$ if $t^* \geq t_0$
2. set $t_X \leftarrow t_1$
3. return $st_{t_0,t_1}$
```

Fig. 2. IKEM security games IND-X$_{\text{IKM}}$ (indistinguishability) and OW-X$_{\text{IKM}}$ (one-way). Red text is only for IND-X$_{\text{IKM}}$. Green text is only for IND-CCA.

the **Chall** oracle is thereafter disabled. Finally, **Expose**() first checks that the challenge epoch t^* is less than the lower endpoint t_0 of the interval. If so, it sets $t_X \leftarrow t_1$ and returns st_{t_0,t_1} to the adversary.

For CCA security, there is also the **Dec**(c) oracle, written in green in Figure 2. This oracle uses the current state st_{t_0,t_1} to decrypt c using IKMR.dec() and returns the resulting key k only if $c \neq c^*$.

Given this security game, we now formally define secure IKEM schemes:

Definition 10. *For* X $\in$ {CPA, RCCA}*, an IKEM scheme* IKM *is* $(T, \varepsilon^{\text{ind-x}}_{\text{IKM}})$-secure *(resp.* $(T, \varepsilon^{\text{ow}}_{\text{IKM}})$-secure*) if for all adversaries $\mathcal{A}$ playing the security game* IND-X$_{\text{IKM}}$ *(resp.* IND$_{\text{OW}}$*) of Figure 2 and running in time T:*

$$\Pr[b \leftarrow_\$ \text{IND-X}_{\text{IKM}}(\mathcal{A})] \leq 1/2 + \varepsilon^{\text{ind-x}}_{\text{IKM}} \qquad (\textit{resp.}\ \Pr[k_0 \leftarrow_\$ \text{OW}_{\text{IKM}}(\mathcal{A})] \leq \varepsilon^{\text{ow}}_{\text{IKM}}).$$

Lower Bound on Secret State Size Unfortunately, we prove a lower bound that shows the size of the secret state of any IKEM scheme must be proportional to the current interval size, $t_1 - t_0$ (times the security parameter, λ). As we write in Section 1, one way to show this would be to prove that IKEM implies a different and simpler symmetric-key primitive, called *Self Encrypted Queue* introduced by Choi et al. [CDV21], for which they also prove a corresponding lower bound. Nevertheless, we provide in Theorem 6 of Appendix B a direct lower bound proof showing that the IKEM secret state size must be $\Omega(\lambda \cdot (t_1 - t_0))$,

```
IKM.gen(1^λ):
    1. generate $(sk, pk) \leftarrow_\$ \mathrm{KM.gen}(1^\lambda)$
    2. set $t_0, t_1 \leftarrow 0, ST[\cdot] \leftarrow \perp$
    3. store $ST[t_1] \leftarrow sk$
    4. return $((ST, t_0, t_1), (pk, t_1))$

IKM.re-gen((ST, t_0, t_1)):
    1. generate $(sk, pk) \leftarrow_\$ \mathrm{KM.gen}(1^\lambda)$
    2. increment $t_1 \leftarrow t_1 + 1$
    3. set $ST[t_1] \leftarrow sk$
    4. return $((ST, t_0, t_1), (pk, t_1))$

IKM.del((ST, t_0, t_1), ℓ):
    1. for $t \in [t_0, t_0 + \ell - 1]$: delete $ST[t]$
    2. set $t_0 \leftarrow t_0 + \ell$
    3. return $(ST, t_0, t_1)$

IKM.enc((pk, t_1), m):
    1. encapsulate $(k, c) \leftarrow_\$ \mathrm{KM.enc}(pk)$
    2. return $(k, (c, t_1))$

IKM.dec((ST, t_0, t_1), (c, t)):
    1. decapsulate $k \leftarrow \mathrm{KM.dec}(ST[t], c)$
    2. return $k$
```

Fig. 3. Base IKEM construction.

as it more clearly illustrates the intuition behind why this is the case. Moreover, the lower bound holds for the IKEM notion with re-encapsulations.

3.2 Optimal Construction from KEM

Given the lower bound of Appendix B, we now provide an IKM construction with secret state size matching the lower bound. The construction is based on a generic KEM scheme, KM and is presented in Figure 3. For each new epoch, the scheme samples a new KM key pair (sk, pk), saves sk to its state, and outputs pk. IKM.enc simply runs the KM encapsulation algorithm on the latest public key pk, and returns the corresponding key k and ciphertext c (appended with the current epoch t_1). IKM.dec then finds the appropriate KM secret key sk, runs the KM decapsulation algorithm and returns the key k.

Now, we show that the IKM scheme of Figure 3 is secure. Intuitively, this is because any (allowed) exposure of the secret state will not leak the KM secret key of the challenge epoch (as it will be deleted from the state), and thus security will follow from that of KM. Moreover, CCA security will be trivially implied by that of KM. The proof of the following Theorem is provided in Appendix D.

Theorem 1. *Let* $X \in \{\mathrm{CPA}, \mathrm{CCA}\}$ *and* KM *be a* $(T, \varepsilon^{\text{ind-x}}_{\mathrm{KM}})$*-secure KEM scheme in the* IND-X *KEM security game. Then the* IKM *construction of Figure 3 is correct and* $(T', T \cdot \varepsilon^{\text{ind-x}}_{\mathrm{KM}})$*-secure, for* $T' \approx T$*, in game* $\text{IND-X}_{\mathrm{IKM}}$ *of Figure 2.*

4 Interval KEM with Re-Encapsulations

In this section, we introduce our extended Interval Key-Encapsulation Mechanism with Re-Encapsulations (IKEMR) notion. We provide a security definition with a decryption oracle and Replayable CCA-style security,[6] but also, by simply removing the decryption oracle, provide a CPA-style security notion. We again note that the same lower bound which shows that the secret state of IKEM must be large also applies to the secret state of IKEMR. Later, we will present two different constructions for this IKEMR notion that provide incomparable efficiency properties. We begin by defining the IKEMR notion:

Syntax An *Interval Key-Encapsulation Mechanism with Re-Encapsulations* (IKEMR) scheme IKMR is a tuple of algorithms IKMR = (IKMR.gen, IKMR.enc, IKMR.dec, IKMR.re-gen, IKMR.del, IKMR.re-enc) with the following syntax:

- IKMR.gen$(1^\lambda) \rightarrow_\$ (st_{0,0}, pk_0)$ generates a secret state and a corresponding public key for interval $[t_0, t_1]$ with $t_0 = t_1 = 0$.
- IKMR.re-gen$(st_{t_0,t_1}) \rightarrow_\$ (st_{t_0,t_1+1}, pk_{t_1+1})$ updates the secret state st_{t_0,t_1} to st_{t_0,t_1+1}, and outputs fresh public key pk_{t_1+1}; i.e., starting new epoch t_1+1 and setting $t_1 \leftarrow t_1+1$.
- IKMR.del$(st_{t_0,t_1}, \ell) \rightarrow st_{t_0+\ell,t_1}$ on input secret state st_{t_0,t_1}, deletes from the secret state the material needed to decapsulate keys encapsulated in the epochs $[t_0, t_0+\ell)$ and outputs $st_{t_0+\ell,t_1}$; i.e., setting $t_0 \leftarrow t_0+\ell$.
- IKMR.enc$(pk_{t_1}) \rightarrow_\$ (k, c)$ on input public key pk_{t_1}, encapsulates key k in ciphertext c.
- IKMR.dec$(st_{t_0,t_1}, c) \rightarrow k$ on input secret state st_{t_0,t_1} and ciphertext c, decapsulates key k.
- IKMR.re-enc$(pk_{t_1}, c) \rightarrow_\$ c'$ re-encapsulates input ciphertext c with respect to the input public key pk_{t_1}.

Correctness An IKEMR scheme is *correct* if secret state st_{t_0,t_1} can decapsulate correctly any ciphertext that was *originally* created in any epoch $t \in [t_0, t_1]$. In particular, even if some ciphertext is re-encapsulated during epoch $t' \in [t_0, t_1]$, if the epoch in which the ciphertext was *originally* created (i.e., when IKMR.enc was executed) is $t < t_0$, then no correctness is required. In fact, as we will see below, such ciphertexts must be secure even given st_{t_0,t_1}. More formally:

Definition 11. *Given $T \in \mathbb{N}$, and dictionary D s.t. $D[i] = (t^i, \ell_i)$ for $i \in [m]$ containing items in $[T] \times [T]$ s.t. $0 < t^1 \leq \cdots \leq t^m \leq T$ and $\sum_{j=1}^{i} \ell_j \leq t^i$ for every $i \in [m]$: let $t_0 \leftarrow 0$; $(st_{0,0}, pk_0) \leftarrow_\$$ IKMR.gen(1^λ); and for all $t_1 \in [T]$, $(st_{t_0,t_1}, pk_{t_1}) \leftarrow_\$$ IKMR.re-gen(st_{t_0,t_1-1}) and for every $i \in [m]$ s.t. for $(t^i, \ell_i) \leftarrow D[i]$, $t^i = t_1$, $st_{t_0+\ell_i,t_1} \leftarrow$ IKMR.del(st_{t_0,t_1}, ℓ_i). Scheme* IKMR *is* correct *if for every such T and D, as well as $r \leq t_1 - t_0 + 1$, $R = \{t_\rho^1, \ldots, t_\rho^r\} \subseteq [t_0, t_1]$ s.t. $t_\rho^1 < \cdots < t_\rho^r$, $(k, c_1) \leftarrow_\$$ IKMR.enc$(pk_{t_\rho^1})$, and for $i \in [2, r]$, $c_i \leftarrow_\$$ IKMR.re-enc$(pk_{t_\rho^i}, c_{i-1})$, $k =$ IKMR.dec(st_{t_0,t_1}, c_r).*

[6] See Section 1 and Appendix C for elaboration on this choice, mainly stemming from the problem of handling decryptions of honest re-encapsulations of the challenge ciphertext.

Initialization: Set $t_X^0, t_X^1, t_R, t^* \leftarrow -\infty$, $t_0, t_1 \leftarrow 0$, and $\mathit{pub\text{-}chall} \leftarrow 0$. Then compute $(st_{t_0,t_1}, pk_{t_1}) \leftarrow_\$ \mathrm{IKMR.gen}(1^\lambda)$ and output pk_{t_1}.

Re-Gen():

```
1. increment $t_1 \leftarrow t_1 + 1$
2. regenerate $(st_{t_0,t_1}, pk_{t_1}) \leftarrow_\$$
     $\mathrm{IKMR.re\text{-}gen}(st_{t_0,t_1-1})$
3. return $pk_{t_1}$
```

Chall(*pub*):

```
1. return $\bot$ if $pub = 1$ and $t_1 = t_X^1$
2. set $t_R, t^* \leftarrow t_1$
3. encapsulate $(k_0, c^*) \leftarrow_\$$
     $\mathrm{IKMR.enc}(pk_{t_1})$
4. sample random $k_1 \leftarrow_\$ \mathcal{K}$
5. flip random coin $b \leftarrow_\$ \{0,1\}$
6. disable Chall()
7. if $pub = 1$ set $\mathit{pub\text{-}chall} \leftarrow 1$
     and return $((k_b, k_{1-b}), c^*)$
```

Expose():

```
1. return $\bot$ if $\mathit{pub\text{-}chall} = 1$ and $t^* \geq t_0$
2. if $t_1 > t_X^1$: set $t_X^0 \leftarrow t_0, t_X^1 \leftarrow t_1$
3. return $st_{t_0,t_1}$
```

Re-Enc-Chall(*pub*):

```
1. return $\bot$ if
     (a) $\mathit{pub\text{-}chall} = 1$; or
     (b) $t^* = -\infty$; or
     (c) $t_1 = t_R$; or
     (d) $pub = 1$, $t_1 = t_X^1$ and $t^* \geq t_X^0$
2. set $t_R \leftarrow t_1$
3. re-encapsulate $c^* \leftarrow_\$$
     $\mathrm{IKMR.re\text{-}enc}(pk_{t_1}, c^*)$
4. if $pub = 1$: set $\mathit{pub\text{-}chall} \leftarrow 1$
     and return $((k_b, k_{1-b}), c^*)$
```

Dec(c):

```
1. decapsulate $k \leftarrow \mathrm{IKMR.dec}(st_{t_0,t_1}, c)$
2. return $\bot$ if $k \in \{k_0, k_1\}$
3. return $k$
```

Del(ℓ):

```
1. return $\bot$ if $t_0 + \ell > t_1$
2. increment $t_0 \leftarrow t_0 + \ell$
3. compute $st_{t_0,t_1} \leftarrow$
     $\mathrm{IKMR.del}(st_{t_0-\ell,t_1}, \ell)$
```

Fig. 4. IKEMR IND-X$_{\mathrm{IKMR}}$ security game, for X $\in \{\mathrm{CPA}, \mathrm{RCCA}\}$. Components only needed for the IND-RCCA$_{\mathrm{IKMR}}$ security game are written in green.

Security Now we define security for an IKEMR scheme. At a high level, this security (i) allows for the challenge ciphertext encrypted in epoch t^* to not be made public (i.e., unavailable to the adversary by setting $pub = 0$) immediately, (ii) allows for re-encapsulations of the challenge ciphertext before it is made public, and (iii) is required if and only if once the challenge ciphertext is made public (via $pub = 1$) after a re-encapsulation in epoch t_1, for every state $st_{t_0',t_1'}$ that the adversary had exposed before the time of publication, either $t^* < t_0'$ or $t_1 > t_1'$ (and also the adversary waits until $t_0 > t^*$ before exposing the state again). In particular, the last condition of the last item implies that even if the adversary earlier exposed multiple states $st_{t_0',t_1'}$ such that $t^* \in [t_0', t_1']$, *before the challenge ciphertext was made public*, then if the receiver re-generates its state to output a fresh public key, is not exposed again until after $t_0 > t^*$, and the challenge ciphertext is re-encrypted with respect to the above public key, the new ciphertext is required to be secure. Our indistinguishability notion is mildly atypical as, once the challenge ciphertext is made public, the adversary is given either (with equal probability 1/2) (i) the real key k_0 encapsulated by the challenge ciphertext, followed by a random key k_1, i.e., (k_0, k_1); or (ii) (k_1, k_0). The adversary must guess if they are in world (i) or (ii). Additionally, for Replayable CCA (RCCA) security, the adversary is allowed to see decryptions of ciphertexts of its choosing, as long as they do not decrypt to k_0 or k_1. In Appendix C, we provide justification for this slightly modified RCCA definition; e.g., it implies RCCA-secure IPKER.

More formally, we define the IND-$\mathrm{X}_{\mathrm{IKMR}}$ ($\mathrm{X} \in \{\mathrm{CPA}, \mathrm{RCCA}\}$) security game for IKEMR in Figure 4. The security game starts by sampling a key pair via IKMR.gen and returning the public key to the adversary. It also initializes a number of variables, including $t_X^0, t_X^1 \leftarrow -\infty$, which will be used to store the endpoints of the latest state $st_{t_X^0, t_X^1}$ that the adversary exposed (these endpoints are only updated when $t_1 > t_X^1$). In addition, the game keeps track of the latest epoch t_R in which the challenge ciphertext was (re-)encapsulated, as well as *pub-chall* which indicates if the challenge ciphertext has been made public (set to 1 if so). Oracle **Re-Gen**() re-generates the state using IKMR.re-gen() and returns the new public key. **Del**(ℓ) deletes old key material for the last ℓ epochs from the state using IKMR.del($\cdot, \ell$), only if the updated interval would still be valid; i.e., $t_0 + \ell \leq t_1$. Oracle **Chall**(*pub*), if $pub = 1$, first checks if the receiver's state has not been exposed in the current epoch. If not, it runs IKMR.enc() to obtain challenge ciphertext c^* and encapsulated key k_0, then samples random k_1 and bit b, and finally returns $((k_b, k_{1-b}), c^*)$. If $pub = 0$, then no matter what, **Chall**() runs IKMR.enc(), but does not output the challenge ciphertext c^* or keys (k_0, k_1); only stores them as well as the challenge epoch t^* and $t_R \leftarrow t^*$. In both cases, the **Chall** oracle is thereafter disabled. Oracle **Expose**() always returns the current state if the challenge ciphertext has not been made public yet (*pub-chall* $= 0$). Otherwise, if the challenge ciphertext is public, **Expose**() first checks that the challenge epoch t^* is not at least t_0, the lower endpoint for which the current state remembers key material, since then the adversary can trivially decapsulate the challenge ciphertext and win the game.

Finally, the re-encapsulation oracle: **Re-Enc-Chall**(*pub*). This oracle first returns $\perp$ if (i) the challenge ciphertext has already been made public (*pub-chall* $=$ 1), (ii) the challenge ciphertext has not already been created ($t^* = -\infty$), or (iii) **Re-Gen**() has not been queried since the last **Chall**() or **Re-Enc-Chall**() query. Now, if $pub = 0$, and the above checks have passed, then **Re-Enc-Chall**() runs $c_1^* \leftarrow_\$$ IKMR.re-enc($\cdot, c_0^*$), stores the re-encapsulated ciphertext c_1^*, and updates $t_R \leftarrow t_1$ to the current epoch, but does not output anything. If $pub = 1$, then in addition to the above checks, **Re-Enc-Chall**() aborts if for the latest state $st_{t_X^0, t_X^1}$ that the adversary exposed, $t_X^1 = t_1$ (the current epoch) and *original* challenge epoch $t^* \geq t_X^0$, since then the adversary could trivially decapsulate the challenge ciphertext and win the game. Once the checks have passed, **Re-Enc-Chall**() runs $c_1^* \leftarrow_\$$ IKMR.re-enc($\cdot, c_0^*$), sets *pub-chall* $\leftarrow 1$ and outputs $((k_b, k_{1-b}), c_1^*)$, where k_0, k_1 are the real and random keys.

For RCCA security, there is also the **Dec**(c) oracle, written in green in Figure 4. This oracle uses the current state st_{t_0,t_1} to decrypt c using IKMR.dec() and returns the resulting key k only if $k \notin \{k_0, k_1\}$. Given this security game, we now formally define secure IKEMR schemes:

Definition 12. *For* $\mathrm{X} \in \{\mathrm{CPA}, \mathrm{RCCA}\}$*, an IKEMR scheme* IKMR *is* $(T, \varepsilon^{\text{ind-x}}_{\text{IKMR}})$-secure *if for all adversaries* $\mathcal{A}$ *playing the security game* $\text{IND-X}_{\text{IKMR}}$ *and running in time* T*:* $\Pr[b \leftarrow_{\$} \text{IND-X}_{\text{IKMR}}(\mathcal{A})] \leq 1/2 + \varepsilon^{\text{ind-x}}_{\text{IKMR}}$.

5 IKEMR Construction from Lossy TDPs with Common Domain

We now present our IKEMR construction from Lossy TDPs with Common Domain that matches the lower bound on secret state size mentioned above. This construction optimizes for small ciphertexts, as small as $O(\lambda + \log(T))$ bits, where T is the total number of epochs, even after many re-encryptions. We provide a RCCA-secure construction that additionally makes use of all-but-one trapdoor functions and one-time signatures (based on a technique of [PW08]). We also demonstrate that by simply removing the ABO and OTS, we obtain a construction that is CPA-secure.

As an intermediate building block, we first define *f-Bounded Forward-Secure Lossy Trapdoor Permutations with Common Domain* and instantiate it using Lossy TDPs with Common Domain.

5.1 *f*-Bounded Forward-Secure Lossy Trapdoor Permutation with Common Domain.

We now introduce f-Bounded Forward-Secure Lossy TDPs (FS-TDPs) with Common Domain. This primitive is similar to f-bounded Forward-Secure KEMs [CHK03], except we require security properties corresponding to Lossy TDPs, instead of KEMs.

Syntax A family of *f-Bounded Forward-Secure Lossy TDPs with Common Domain* (FSP) is a tuple of algorithms FSP = (FSP.gen, FSP.up, FSP.eval, FSP.inv) with the following syntax:

– $\text{FSP.gen}(1^\lambda, f, b, t^*) \rightarrow_{\$} (SK_0, PK)$ on inputs $b \in \{0,1\}$ and t^* that specify whether the generated instance is *always-injective* ($b = 1$) or *lossy* in epoch t^* ($b = 0$), outputs initial secret key SK_0 and public key PK.
– $\text{FSP.up}(SK_t) \rightarrow SK_{t+1}$ updates input secret key SK_t to SK_{t+1}.
– $\text{FSP.eval}(PK, t', x) \rightarrow y$ on input PK and epoch t', evaluates the permutation for epoch t' on x to give output y.
– $\text{FSP.inv}(SK_t, t', y) \rightarrow x$ on input SK_t and t', inverts the permutation for epoch t' on y to give x.

Note that t' is an explicit input of FSP.inv(). For our use of FSP.inv() within our IKEMR construction, we will be able to extract t' from the rest of the ciphertext.

Correctness Correctness requires that in *always-injective* mode, for $t_0 \le t_1$, the receiver with SK_{t_0} can invert $y \leftarrow \text{FSP.eval}(PK, t_1, x)$ to x.

Definition 13. *Scheme* FSP *is* correct *if for all* $(SK_0, PK) \leftarrow_\$ \text{FSP.gen}(1^\lambda, f, 1, t^*)$*, for every* $t \in [f-1], SK_t \leftarrow \text{FSP.up}(SK_{t-1})$*, all* $x \in \mathcal{X}$ *and any* $0 \le t_0 \le t_1 \le f-1$*,* $x = \text{FSP.inv}(SK_{t_0}, t_1, \text{FSP.eval}(PK, t_1, x))$*.*

Security For *lossy* mode with respect to given epoch $t^* < f$, we require that the size of the image of the evaluation algorithm with respect to epoch t^* is much smaller than the domain $\mathcal{X}$:

Definition 14. *Scheme* FSP *is* L-lossy *if for all* $(SK_0, PK) \leftarrow_\$ \text{FSP.gen}(1^\lambda, f, 0, t^*)$*,* $|\text{FSP.eval}(PK, t^*, \cdot)| \le |\mathcal{X}|/L$*, where* $|\text{FSP.eval}(PK, t^*, \cdot)|$ *is the number of different outputs across all inputs* $x \in \mathcal{X}$*.*

Furthermore, for any adversary with only the public key PK and *any* secret key $SK_{t'}$ for $t' > t^*$, we require that it is hard to distinguish whether they were sampled in always-injective mode or lossy mode with respect to t^* (even with t^* known to the adversary):

Definition 15. *Scheme* FSP *is* $(T, \varepsilon_{\text{FSP}})$-secure *if for all adversaries* $\mathcal{A}$ *running in time* T*:*

$$\begin{aligned}\Pr[b \leftarrow_\$ \mathcal{A}(SK_{t'}) : b \leftarrow_\$ \{0,1\}; (0 \le t^* \le f-1) \leftarrow_\$ \mathcal{A}();\\ (SK_0, PK) \leftarrow_\$ \text{FSP.gen}(1^\lambda, f, b, t^*); (t^* < t' \le f) \leftarrow_\$ \mathcal{A}(PK);\\ \textit{for } t \in [f-1], SK_t \leftarrow \text{FSP.up}(SK_{t-1})] \le 1/2 + \varepsilon_{\text{FSP}}\end{aligned}$$

FSP Construction We now provide a simple construction based on a family of Lossy TDPs with Common Domain P and PRG G. At a high level, FSP.gen will generate f instantiations of P: for epoch t^* and lossiness bit b, the t^*-th instantiation of P will be generated with bit b; for all other epochs t, the t-th instantiation of P will be generated with bit 1 (i.e., injective). The initial secret key SK_0 will store a PRG s_0 which can be expanded deterministically (in a chain) to sample f secret keys and the public key PK will store the f corresponding public keys. To update secret key SK_t, the receiver simply expands the seed s_t and deletes the output secret key corresponding to the t-th instantiation of P (while still s_{t+1} can be expanded to compute all future secret keys). To evaluate with respect to epoch t, the sender simply evaluates the t-th instantiation of P on x. Finally, to invert using SK_t on input y and epoch t', the receiver simply expands s_t iteratively to get sk'_t corresponding to the t'-th instantiation of P and uses it to invert y. The scheme is as follows:

- FSP.gen$(1^\lambda, f, b, t^*)$: set $t \leftarrow 0$ and sample random $s_0 \leftarrow_\$ \mathcal{S}$. For $i \in [0, f-1] \backslash \{t^*\}$, compute $(s_{i+1}, r_i) \leftarrow \text{G}(s_i)$ and sample $(sk_i, pk_i) \leftarrow_\$ \text{P.gen}(1^\lambda, 1; r_i)$. For t^*, compute $(s_{t^*+1}, r_{t^*}) \leftarrow \text{G}(s_{t^*})$ and sample $(sk_{t^*}, pk_{t^*}) \leftarrow_\$ \text{P.gen}(1^\lambda, b; r_{t^*})$. Set $SK_t \leftarrow (t, s_0)$ and output $PK \leftarrow \{pk_0, \ldots, pk_{f-1}\}$.
- FSP.up(SK_t): Compute $(s_{t+1}, \cdot) \leftarrow \text{G}(s_t)$ and set $t \leftarrow t+1$.

– FSP.eval(PK, t, x): Compute and output $y \leftarrow \mathrm{P.eval}(pk_t, x)$.
– FSP.inv(SK_t, t', y): For i from t to t': compute $(s_{i+1}, r_i) \leftarrow_\$ \mathrm{G}(s_i)$. Then sample $(sk_{t'}, \cdot) \leftarrow_\$ \mathrm{P.gen}(1^\lambda, 1; r_{t'})$ and output $x \leftarrow \mathrm{P.inv}(sk_{t'}, y)$.

We now show that the above FSP construction is correct, L-lossy, and secure. The correctness and L-lossiness clearly follows from that of P. Furthermore, for security, since the FSP secret key $SK_{t'}$ that the adversary receives will not contain the lossy epoch t^*'s secret key sk_{t^*} of the lossy TDP family P (as it will have been deleted), but only a random PRG seed past epoch t^* in the chain, security follows directly from that of P and G. The proof of the following Theorem is provided in Appendix D.

Theorem 2. *If* P *is correct, L-lossy, and $(T, \varepsilon_\mathrm{P})$-secure, and* G *is $(T, \varepsilon_\mathrm{G})$-secure then the above* FSP *construction is correct, L-lossy, and $(T', \varepsilon_\mathrm{P} + T \cdot \varepsilon_\mathrm{G})$-secure, for $T' \approx T$.*

5.2 IKMR Construction

Given the FSP primitive, we can now present our construction for IKEMR. The construction IKMR$_\varDelta$ is formally presented in Figure 5. It is parameterized by an (efficiently computable) function $\varDelta : \mathbb{N} \to \mathbb{N}$, such that for each input T, $\varDelta(T) \leq T$. Intuitively, if the size of the active interval $t_1 - t_0$ stays roughly the same throughout, then the public key size will be proportional to $\varDelta(t_1 - t_0)$ and the size of any ciphertext (regardless of the number of times it's been (re-)encapsulated), will be proportional to $\lambda + \beta \cdot (t_1 - t_0)/\varDelta(t_1 - t_0)$, where $\beta = O(\log \lambda)$ is the number of bits needed to represent each epoch. In particular, if $\varDelta(T) = T$, then all ciphertexts (no matter how many re-encapsulations) will be of size proportional to only λ (since $\beta = O(\log \lambda)$). Moreover, even if $\varDelta(T)$ is small (even $\varDelta(T) = 1$), then the ciphertext grows with (almost) every re-encapsulation, but only by a β factor, *independent* of λ.

In addition to the FSP primitive, our construction IKMR$_\varDelta$ utilizes a family $\mathcal{H}$ of pairwise independent hash functions from $\{0,1\}^n \to \{0,1\}^\ell$. For RCCA security, $\ell \leq k - 2\log(1/\varepsilon_\mathcal{H})$, for some $k = \omega(\log n)$ and negligible $\epsilon_\mathcal{H} = \mathsf{negl}(\lambda)$; for CPA security, $\ell \leq \log(L) - 2\log(1/\epsilon_\mathcal{H})$, where L the lossiness of FSP. Additionally for RCCA security, we make use of an all-but-one trapdoor function family ABO and one-time signature scheme OTS.

In IKMR$_\varDelta$.gen, the receiver samples random hash function $h \leftarrow_\$ \mathcal{H}$, generates an FSP instance $(sk_1, pk_1) \leftarrow_\$ \mathrm{FSP.gen}(1^\lambda, t_1 - t_0 + 1, 1, \perp)$, sets $t_0, t_1 \leftarrow 1$, and sets $(SK[t_1], PK[t_1]) \leftarrow ((sk_1, t_0), (pk_1, t_0))$. We will explain the choice of instantiating FSP with $f = t_1 - t_0 + 1$ while explaining IKMR$_\varDelta$.re-gen and IKMR$_\varDelta$.del below. Additionally, for RCCA security, the receiver samples an ABO instance $(\cdot, pk) \leftarrow_\$ \mathrm{ABO.gen}(1^\lambda, 0^v)$, where v is the bit-length of verification keys generated by OTS (ABO is only needed for security and indeed the secret key of the ABO is not used by the receiver). For each IKMR$_\varDelta$.re-gen execution, the receiver increments $t_1 \leftarrow t_1 + 1$, samples a new FSP instance $(sk_{t_1}, pk_{t_1}) \leftarrow_\$ \mathrm{FSP}(1^\lambda, t_1 - t_0 + 1, 1, \perp)$, sets $(SK[t_1], PK[t_1]) \leftarrow ((sk_{t_1}, t_0), (pk_{t_1},$

```
$\underline{\mathrm{IKMR}_\Delta.\mathrm{gen}(1^\lambda)}$:

1. set $t_0, t_1 \leftarrow 1$, $SK[\cdot], PK[\cdot] \leftarrow \perp$
2. sample $h \leftarrow_\$ \mathcal{H}$
3. generate $(sk, pk) \leftarrow_\$$
     $\mathrm{FSP.gen}(1^\lambda, 1, 1, \perp)$
4. generate $(\cdot, pk') \leftarrow_\$$
     $\mathrm{ABO.gen}(1^\lambda, 0^v)$
5. set $(SK[1], PK[1]) \leftarrow$
     $((sk, t_0), (pk, t_0))$
6. return $((SK, PK, pk', h, t_0, t_1),$
     $(PK, pk', h, t_0, t_1))$

$\underline{\mathrm{IKMR}_\Delta.\mathrm{del}((SK, PK, pk', h, t_0, t_1), \ell)}$:

1. for $i \in [\ell]$:
   (a) set $SK[t_0 + i - 1] \leftarrow \perp$
   (b) for $t \in [t_0 + \ell, t_1]$:
         set $(sk, t') \leftarrow SK[t]$;
         update $sk' \leftarrow \mathrm{FSP.up}(sk)$;
         store $SK[t] \leftarrow (sk', t')$
2. increment $t_0 \leftarrow t_0 + \ell$
3. return $(SK, PK, pk', h, t_0, t_1)$

$\underline{\mathrm{IKMR}_\Delta.\mathrm{re\text{-}enc}((PK, pk', h, t_0, t_1),}$
    $\underline{(c_1, vk, c_2, \sigma,}$
    $\underline{((t_{0,0}, t_{0,1}), \dots, (t_{l,0}, t_{l,1}))))}$:
1. if $t_0 > t_{0,0}$ return $\perp$
2. if $PK[t_{l,1} + 1] \neq \perp$:
   (a) for $t'$ from $(t_{l,1} + 1)$ to $t_1$:
         let $(pk_{t'}, t'_0) \leftarrow PK[t']$;
         compute $c_1 \leftarrow$
          $\mathrm{FSP.eval}(pk, t_{0,0} - t'_0, c_1)$
   (b) return $(c_1, vk, c_2, \sigma,$
         $((t_{0,0}, t_{0,1}), \dots, (t_{l,0}, t_1)))$
3. else:
   (a) compute $c_1 \leftarrow$
         $\mathrm{FSP.eval}(PK[t_1], t_{0,0} - t_0, c_1)$
   (b) return $(c_1, vk, c_2, \sigma, ((t_{0,0}, t_{0,1}),$
         $\dots, (t_{l,0}, t_{l,1}), (t_1, t_1)))$
```

```
$\underline{\mathrm{IKMR}_\Delta.\mathrm{re\text{-}gen}((SK, PK, pk', h, t_0, t_1))}$

1. increment $t_1 \leftarrow t_1 + 1$
2. generate $(sk, pk) \leftarrow_\$$
     $\mathrm{FSP.gen}(1^\lambda, t_1 - t_0 + 1, 1, \perp)$
3. set $(SK[t_1], PK[t_1]) \leftarrow$
     $((sk, t_0), (pk, t_0))$
4. set
     $PK[\leq t_1 - \Delta(t_1 - t_0) - 1] \leftarrow \perp$
5. return $((SK, PK, pk', h, t_0, t_1),$
     $(PK, pk', h, t_0, t_1))$

$\underline{\mathrm{IKMR}_\Delta.\mathrm{enc}((PK, pk', h, t_0, t_1))}$:

1. sample random $x \leftarrow_\$ \mathcal{X}$
2. compute $c_1 \leftarrow$
     $\mathrm{FSP.eval}(PK[t_1], t_1 - t_0, x)$
3. generate $(sk, vk) \leftarrow_\$ \mathrm{OTS.gen}(1^\lambda)$
4. compute $c_2 \leftarrow \mathrm{ABO.eval}(pk', vk, x)$
5. sign $\sigma \leftarrow \mathrm{OTS.sign}(sk, c_2)$
6. return $(h(x), (c_1, vk, c_2, \sigma, ((t_1, t_1))))$

$\underline{\mathrm{IKMR}_\Delta.\mathrm{dec}((SK, PK, pk', h, t_0, t_1),}$
    $\underline{(c_1, vk, c_2, \sigma,}$
    $\underline{((t_{0,0}, t_{0,1}), \dots, (t_{l,0}, t_{l,1}))))}$:
1. return $\perp$ if $\mathrm{OTS.ver}(vk, c_2, \sigma) = 0$
2. set $c'_1 \leftarrow c_1$
3. for $i$ from $l$ to $0$:
   (a) for $t'$ from $t_{i,1}$ to $t_{i,0}$: let
         $(sk_{t'}, t'_0) \leftarrow SK[t']$; invert
         $c'_1 \leftarrow \mathrm{FSP.inv}(SK[t'], t_{0,0} - t'_0, c'_1)$
4. set $x \leftarrow c'_1$
5. for $i$ from $0$ to $l$
   (a) for $t'$ from $t_{i,0}$ to $t_{i,1}$: let
         $(pk_{t'}, t'_0) \leftarrow PK[t']$;
         compute $c'_1 \leftarrow$
          $\mathrm{FSP.eval}(PK[t'], t_{0,0} - t'_0, c'_1)$
6. return $\perp$ if $c'_1 \neq c_1$
7. compute $c'_2 \leftarrow \mathrm{ABO.eval}(pk', vk, x)$
8. return $\perp$ if $c'_2 \neq c_2$
9. compute and return $h(x)$
```

Fig. 5. TDP-based IKEM with Re-Encapsulations construction. Text written in green is only needed for IND-RCCA$_{\mathrm{IKMR}}$ security.

$t_0)$), and deletes from PK all entries except those for the latest $\Delta(t_1 - t_0) + 1 \leq t_1 - t_0 + 1$ epochs. The reason we instantiate the FSP with $f = t_1 - t_0 + 1$ is that we will use the FSP epoch $t' \in [0, f-1]$ to (re-)encapsulate IKEMR ciphertexts originally created in IKEMR epoch $t_0 + t' \in [t_0, t_1]$ (explanation continues after IKMR$_\Delta$.del below). Then in IKMR$_\Delta$.del(ℓ), the receiver simply deletes from the secret key SK the FSP keys $sk_{t_0}, \ldots, sk_{t_0+\ell-1}$, and updates all other FSP keys $sk_{t_0+\ell}, \ldots, sk_{t_1}$, ℓ times each. Therefore, even if IKMR$_\Delta$.del(ℓ_i) is called m times such that $\sum_{i=1}^{m} \ell_i \leq t'$, the lower endpoint of the active interval is moved from t_0 to at most $t_0 + t'$. Thus, the FSP instance sampled when the lower endpoint was t_0, having been updated at most t' times will still be able to decapsulate IKEMR ciphertexts originally created in IKEMR epoch $t_0 + t'$. However, if $\sum_{i=1}^{m} \ell_i > t'$, moving the lower endpoint of the active interval from t_0 to after $t_0 + t'$, then the FSP instance sampled when the lower endpoint was t_0, having been updated more than t' times will no longer be able to decapsulate IKEMR ciphertexts originally created in IKEMR epoch $t_0 + t'$, as required by security.

Thus, to encapsulate in epoch t_1, IKMR$_\Delta$.enc samples random $x \leftarrow_\$ \{0,1\}^n$, then evaluates the FSP for public key pk_{t_1} on x and the FSP instantiation's epoch $t_1 - t_0$ to get output c_1. For RCCA security, the encapsulator also (i) samples OTS key pair (sk, vk); (ii) evaluates the ABO on branch vk and input x to obtain output c_2; and (iii) finally signs c_2 using OTS with signing key sk to obtain σ. The output key is $h(x)$ and the ciphertext is (c_1, vk, c_2, σ) appended with tuple (t_1, t_1). Note that in the RCCA-secure construction, vk, c_2, σ will remain untouched in the ciphertext even after re-encapsulations.

To re-encapsulate ciphertext $(c_1, vk, c_2, \sigma, ((t_{0,0}, t_{0,1}), \ldots, (t_{l-1,0}, t_{l-1,1}), (t_{l,0}, t_{l,1})))$ in epoch t_1 on input public key with active interval $[t_0, t_1]$, $t_{0,0}$ is interpreted as the epoch in which the ciphertext was originally created. Thus, IKMR$_\Delta$.re-enc first returns $\perp$ if $t_0 > t_{0,0}$ since if this is the case, then the receiver must not be able to decapsulate the ciphertext anyway, as required for security. For the RCCA-secure construction, the re-encapsulator also keeps vk, c_2, σ untouched in the eventually output ciphertext. Then, IKMR$_\Delta$.re-enc checks if there is an FSP public key in PK for epoch $t_{l,1} + 1$. If so, then for t' from $t_{l,1} + 1$ to t_1: IKMR$_\Delta$.re-enc first retrieves $(pk_{t'}, t'_0) \leftarrow PK[t']$, where t'_0 was the lower endpoint of the active interval *when epoch t' was created.* Next IKMR$_\Delta$.re-enc evaluates the FSP for public key $pk_{t'}$ on c_1 and the FSP instantiation's epoch $t_{0,0} - t'_0$, as specified above, to get new output c'_1. Then, IKMR$_\Delta$.re-enc outputs the same ciphertext as above, except c_1 replaced by the final output c'_1 and $t_{l,1}$ of the final evaluation epoch tuple replaced with t_1. The latter is because each of these tuples represent the evaluation intervals of epochs t' in which the FSP for the corresponding public key $pk_{t'}$ was evaluated on c.

If there is no FSP public key in PK for epoch $t_{l,1} + 1$, then IKMR$_\Delta$.re-enc just evaluates the FSP for public key pk_{t_1} of epoch t_1 on c_1 and the FSP instantiation's epoch $t_{0,0} - t_0$, as specified above, to get new output c'_1. Then, IKMR$_\Delta$.re-enc outputs the same ciphertext as above, except c_1 replaced by c'_1 and (t_1, t_1) appended to the list of evaluation epoch tuples (because we have started a new evaluation interval).

Finally, to decapsulate ciphertext $(c_1, vk, c_2, \sigma, ((t_{0,0}, t_{0,1}), \ldots, (t_{l-1,0}, t_{l-1,1}), (t_{l,0}, t_{l,1})))$, for each tuple $(t_{i,0}, t_{i,0})$ for i from l to 0, sequentially for t' from $t_{i,1}$ to $t_{i,0}$, the receiver first retrieves $(sk_{t'}, t'_0) \leftarrow SK[t']$, where t'_0 was the lower endpoint of the active interval *when epoch t' was created*. Then, IKMR_Δ.dec inverts c_1 using the FSP secret key $sk_{t'}$ with respect to the FSP instantiation's epoch $t_{0,0} - t'_0$ (the same epoch on which it was evaluated). For RCCA-security, the receiver also verifies $\mathrm{OTS.ver}(vk, c_2, \sigma) = 1$ and recomputes c_1 and c_2 to check well-formedness. Then, using the final inverse $x \leftarrow c_1$ from above, IKMR.dec outputs $h(x)$ as the key.

Efficiency Before formally analyzing the security of IKMR_Δ we will provide some bounds on the efficiency of the construction. It is clear that the size of every public key is proportional to $\Delta(t_1 - t_0)$. We will now attempt to bound the size of every (even re-encapsulated) ciphertext. First, the components needed for RCCA security, vk, c_2, σ only add $O(\lambda)$ bits and stay untouched even after several re-encapsulations. Now, for each re-encapsulation during epoch t_1, if the last epoch $t_{l,1}$ in which the ciphertext was (re-)encapsulated is such that $t_{l,1} + 1 \in [t_1 - \Delta(t_1 - t_0) - 1, t_1]$ then PK contains FSP public keys $pk_{t'}$ for $t' \in [t_{l,1} + 1, t_1]$. Therefore, the c part of the ciphertext can be re-evaluated sequentially using the FSP public keys $pk_{t'}$ for all such t', and the last epoch $t_{l,1}$ of the last evaluation interval of the ciphertext can be replaced with t_1. Observe that in this case, the ciphertext *does not grow*, assuming that each epoch can be represented using some fixed $\beta = O(\log \lambda)$ number of bits. Indeed, only if $t_{l,1} + 1 \notin [t_1 - \Delta(t_1 - t_0) - 1, t_1]$ and therefore PK does not contain a FSP public key for $t_{l,1}$, must the ciphertext grow. In this case, IKMR_Δ.re-enc skips to evaluating the c part of the ciphertext on only pk_{t_1}, and appends to the ciphertext evaluation interval (t_1, t_1). Here, the ciphertext grows by $O(\beta)$ bits.

Consider the case that some ciphertext is (re-)encapsulated in epochs $t^1 < \cdots < t^r$ such that for each $i \in [r]$, the active interval when epoch t^i was created was $[t^i_0, t^i_1]$ (where $t^i_1 = t^i$). Then, it is clear that if for any $i \in [r]$, $t^i_0 > t^1$, the ciphertext size becomes 0, as the original creation epoch t^1 is outside of the active interval and thus the re-encryptor sets the ciphertext to $\perp$. Otherwise, let $\delta_i = \Delta(t^i_1 - t^i_0)$ for $i \in [n]$ and let $\delta^*_1, \ldots, \delta^*_r$ be $\delta_1, \ldots, \delta_r$ sorted in ascending order. In Lemma 1 below, we in fact show that the number of times G the ciphertext grows is bounded by

$$G \leq \mathrm{argmax}_g \left\{ t^1 + \sum_{i=1}^{g} \delta^*_i \leq t^r \right\}. \tag{1}$$

Therefore, the ciphertext size is bounded by $O(\lambda + G \cdot \beta)$, where β is the number of bits used to represent each epoch.

Lemma 1. *Given (re-)encapsulation epochs $t^1 < \cdots < t^r$ with active intervals $[t^i_0, t^i_1]$ when created, let $\delta_i = \Delta(t^i_1 - t^i_0)$ for $i \in [r]$. Let $\delta^*_1, \ldots, \delta^*_r$ be $\delta_1, \ldots, \delta_r$ sorted in ascending order. Then, the number of times G the ciphertext can grow is bounded by Equation 1.*

Proof. The ciphertext is first encapsulated at time t^1 and last re-encapsulated at time t^r. Moreover, we know that the j-th re-encryption only grows the ciphertext if $t_1^j - \delta_j > t_1^{j-1}$, or $t_1^j - t_1^{j-1} > \delta_j$. Thus, it must be that for those j s.t. the above is true: $t^1 + \sum_j \delta_j \leq t^r$. Therefore, the maximum number G of such j above corresponds to $\mathrm{argmax}_g\{t^1 + \sum_{i=1}^{g} \delta_i^* \leq t^r\}$. $\square$

Corollary 1. *G is bounded by $(t^r - t^1)/\min_j \delta_j$.*

Corollary 2. *If $(t_1^1 - t_0^1) = \cdots = (t_1^r - t_0^r) = T_1 - T_0$, then the number of times G the ciphertext can grow is bounded by $(t^r - t^1)/\Delta(T_1 - T_0)$.*

Security Now, we show that construction IKMR_Δ of Figure 5 is correct and secure. Before formally stating the theorem, we give some intuition on the security of the scheme. First, recall that the ABO secret key is not stored by the receiver. Now, let t^* be the epoch in which the challenge ciphertext is originally created and let t_{pub} be the epoch in which the (re-)encapsulated challenge ciphertext is made public. By the definition of the security game, it cannot be the case that any st_{t_0,t_1} with $t_0 \leq t^* \leq t_{pub} \leq t_1$ is ever leaked to the adversary. Indeed, it must be that either $t_1 < t_{pub}$ or $t^* < t_0$. In the former case, it is easy to see that *no* information about the FSP secret key $sk_{t_{pub}}$ generated for epoch t_{pub} is leaked to the adversary (beyond $pk_{t_{pub}}$). In the latter case, a version of $sk_{t_{pub}}$ may be leaked to the adversary, but only a version that has been updated past (its internal FSP epoch for corresponding IKMR_Δ) epoch t^*. Therefore, by the security of FSP and ABO, the inverse of the final evaluations c_1, c_2 in the challenge ciphertext can be many possible values, and thus the same can be said about the original random $x \leftarrow_\$ \{0,1\}^n$ sampled for the challenge ciphertext. As a result, from x, a key that is indistinguishable from random is extracted using the pairwise independent hash function h. Moreover, for the RCCA security proof, we are able to always decrypt ciphertexts involving honest (re-)encapsulations using the public key output in epoch t_{pub} by first switching the ABO to lossy mode on branch vk^* before switching the FS-TDP to lossy mode, where vk^* is the sampled verification key for the challenge ciphertext, and then using the ABO to decapsulate instead of the FS-TDP (in the hybrid worlds, the receiver keeps the ABO secret key). Indeed, due to the unforgeability of the OTS scheme, the ABO will never have to invert on branch vk^*, and thus this modified decapsulation will always succeed. The proofs of the following Theorems are provided in Appendix D.

Theorem 3. *Let* FSP *be a family of correct, L-lossy, and $(T, \varepsilon_{\mathrm{FSP}})$-secure trapdoor permutations on common domain $\mathcal{X} = \{0,1\}^n$; let $\mathcal{H}_2 : \{0,1\}^n \to \{0,1\}^\ell$ be a family of pairwise independent hash functions where $\ell \leq k - 2\log(1/\varepsilon_{\mathcal{H}})$, for some $k = \omega(\log n)$ and some negligible $\varepsilon_{\mathcal{H}_2} = \mathsf{negl}(\lambda)$, where $\log(L) + \log(L') \geq n + k$;* OTS *be a strongly unforgeable one-time signature scheme where the verification keys are in $\{0,1\}^v$; and* ABO *be a family of correct, L'-lossy, and $(T, \varepsilon_{\mathrm{ABO}})$-secure all-but-one trapdoor functions on domain $\{0,1\}^n$. Then, for $T' \approx T$, the* IKMR *construction of Figure 5 is correct and $((T', T^2 \cdot (O(1/2^\lambda) + \varepsilon_{\mathrm{OTS}} + 2 \cdot (\varepsilon_{\mathrm{ABO}} + \varepsilon_{\mathrm{FSP}}) + \varepsilon_{\mathcal{H}_2}))$-secure) in game* IND-RCCA$_{\mathrm{IKMR}}$ *of Figure 4.*

Note that given the Lossy TDP of [AKPS19] with $L = 2^{n/4}$ and the ABO of [PW08] with $L' = 2^n/\lambda$, we indeed have that $\log(L)+\log(L') = 5n/4-\log\lambda \geq n + \omega(\lg n)$.

6 IKEMR Construction from FS-PKE

While the advantage of our TDP-based IKEMR construction is the small ciphertext size, a disadvantage is the public key size. Using Forward-Secure Public-Key Encryption (FS-PKE), we can reduce the public key to an element of constant size at the cost of increasing the ciphertext size; furthermore, also the secret key size is slightly increased.

IKMR construction. Since, for our TDP-based IKEMR construction, we already introduce FS-TDP as an abstraction layer, the primary difference towards our FS-PKE-based construction is the omission of parameter Δ. For security against active adversaries, we use an additional collision-resistant hash function. This simplifies the description of our construction from Figure 6 significantly.

Initial key generation via IKMR.gen generates an FS-PKE key pair with one (update-)epoch for the FS-PKE secret key as well as a key for the hash function. Re-generating a key pair via IKMR.re-gen for interval $[t_0, t_1]$ generates an FS-PKE key pair with at most $t_1 - t_0 + 1$ update-epochs; the generated FS-PKE public key becomes the new IKEMR public key and the new FS-PKE secret key is added to the decapsulation interval. Consequently, the IKEMR public key always only consists of a single FS-PKE public key. Deleting l slots from the decapsulation interval via IKMR.del removes the oldest l FS-PKE secret keys entirely; all remaining $t_1 - t_0 - l$ FS-PKE secret keys are updated l times each. This mechanism as well as the underlying rationale resemble those of our FS-TDP-based construction: only decapsulation of ciphertexts initially created after the beginning of the current decapsulation interval should be possible; in particular, ciphertexts encapsulated earlier but re-encapsulated later than that must not be recoverable.

At encapsulation via IKMR.enc, a randomly sampled key k is FS-PKE encrypted to the last update-epoch $t_1 - t_0$ of the current public key; the resulting ciphertext is appended with the current epoch index t_1. For re-encapsulation of ciphertext c with public key PK via IKMR.re-enc, c is FS-PKE encrypted to update-epoch $t_0^\circ - t_0$, where t_0° is the time at which the first version of c was initially created and t_0 was the oldest slot of the decapsulation interval when PK was most recently re-generated. For active security, the a hash of the history of (re-)encapsulation epochs is added to each encrypted payload. Due to the FS-PKE re-encryption as well as the added hash value, the resulting IKEMR ciphertext grows with the number of IKEMR re-encapsulations by the encryption overhead of the FS-PKE scheme as well as the hash length. Additionally, each (re-)encapsulation attaches the current epoch index to the ciphertext, which is used at decapsulation via IKMR.dec to execute the matching FS-PKE decryptions; note this epoch list is only attached once to the outmost encryption layer.

```
IKMR.gen($1^\lambda$):

 1. set $t_0, t_1 \leftarrow 1$, $SK[\cdot], PK \leftarrow \perp$
 2. generate $\kappa \leftarrow_\$ \{0,1\}^\lambda$
 3. generate $(sk, pk) \leftarrow_\$$
      FSE.gen($1^\lambda, 1$)
 4. set $(SK[1], PK) \leftarrow$
      $((sk, t_0), pk)$
 5. return $((\kappa, SK, t_0, t_1),$
      $(\kappa, PK, t_0, t_1))$
```

```
IKMR.re-gen($(\kappa, SK, t_0, t_1)$)

 1. increment $t_1 \leftarrow t_1 + 1$
 2. generate $(sk, pk) \leftarrow_\$$
      FSE.gen($1^\lambda, t_1 - t_0 + 1$)
 3. set $(SK[t_1], PK) \leftarrow$
      $((sk, t_0), pk)$
 4. return $((\kappa, SK, t_0, t_1),$
      $(\kappa, PK, t_0, t_1))$
```

```
IKMR.enc($(\kappa, PK, t_0, t_1)$):

 1. sample random $k \leftarrow_\$ \mathcal{K}$
 2. compute $c \leftarrow_\$$
      FSE.enc($PK, t_1 - t_0, (k, \mathrm{H}_\kappa(t_1))$)
 3. return $(k, (c, (t_1)))$
```

```
IKMR.del($(\kappa, SK, t_0, t_1), \ell$):

 1. for $i \in [\ell]$:
    (a) set $SK[t_0 + i - 1] \leftarrow \perp$
    (b) for $t \in [t_0 + \ell, t_1]$:
          set $(sk, t') \leftarrow SK[t]$;
          update $sk' \leftarrow$ FSE.up($sk$);
          store $SK[t] \leftarrow (sk', t')$
 2. increment $t_0 \leftarrow t_0 + \ell$
 3. return $(\kappa, SK, t_0, t_1)$
```

```
IKMR.re-enc($(\kappa, PK, t_0, t_1), (c, (t_0^\circ, \ldots, t_l^\circ))$):

 1. if $t_0 > t_0^\circ$ return $\perp$
 2. compute $h \leftarrow \mathrm{H}_\kappa((t_0^\circ, \ldots, t_l^\circ, t_1))$
 3. compute
      $c \leftarrow_\$$ FSE.enc($PK, t_0^\circ - t_0, (c, h)$)
 4. return $(c, (t_0^\circ, \ldots, t_l^\circ, t_1))$
```

```
IKMR.dec($(\kappa, SK, t_0, t_1), (c, (t_0^\circ, \ldots, t_l^\circ))$):

 1. for $i$ from $l$ to 0:
    (a) let $(sk_i, t_0') \leftarrow SK[t_i^\circ]$; decrypt
          $(c, h) \leftarrow$ FSE.dec($sk_i, t_0^\circ - t_0', c$)
    (b) if $h \neq \mathrm{H}_\kappa((t_0^\circ, \ldots, t_i^\circ))$: return $\perp$
 2. set $k \leftarrow c$
 3. return $k$
```

Fig. 6. FS-PKE-based IKEM with Re-Encapsulations construction. Text written in green is only needed for IND-RCCA$_{\mathrm{IKMR}}$ security.

Efficiency Clearly, the public key size of this construction is constant for an FS-PKE scheme with constant sized public keys. For clarity, we use the original performance metric by Canetti et al. [CHK03] that is based on the Gentry-Silverberg HIBE [GS02], where the public key is of constant size and the overhead of secret and public key is logarithmic in the total number of update-epochs, respectively. Based on this, our ciphertexts grow linearly in r, which is the number of re-encapsulations. This is increased by the ciphertext overhead of the underlying FS-PKE as well as additional epoch indexes. In total, this yields a ciphertext overhead of $O(r \cdot \log(t_1 - t_0))$. Finally, each secret key in the decapsulation interval is of size $\log(t_1 - t_0)$, which yields a total secret key size of $O((t_1 - t_0) \cdot \log(t_1 - t_0))$.

Security Intuitively, almost the same argument works to prove security of the FS-PKE-based IKEMR construction as for our TDP-based one. For a challenge ciphertext c^* published at time t^*, none of the states exposed at time $t' < t^*$

contains the corresponding FS-PKE secret key. Furthermore, as soon as the initial creation time t_1 of the first version of c^* is deleted from the decapsulation interval, the FS-PKE secret key from time t^* was updated at least $t^* - t_1 + 1$ times. Based on this, instead of embedding a lossy TDP in epoch t^*, we replace the key, respectively ciphertext, that is FS-PKE (re-)encrypted at time t^* with a random string of the same length. Detecting this modification breaks the FS-PKE scheme. For security against active adversaries, we additionally bind the attached list of re-encapsulation epochs via the encrypted hash value. Intuitively, this yields the following Theorem that we formally prove in Appendix D.

Theorem 4. *Let* H *be a* $(T_{\mathrm{H}}, \varepsilon_{\mathrm{H}})$*-collision-resistant hash function and* FSE *be a correct and* $(T_{\mathrm{FSE}}, \varepsilon_{\mathrm{FSE}})$*-secure FS-PKE. Then the* IKMR *construction of Figure 6 is correct and* $(T', \varepsilon_{\mathrm{H}} + T^2 \cdot \varepsilon_{\mathrm{FSE}})$*-secure in game* IND-RCCA$_{\mathrm{IKMR}}$ *of Figure 4, for* $T' \approx T_{\mathrm{H}} + T_{\mathrm{FSE}}$.

Alternative via FS-KEM and AEAD In Appendix E, Figure 9, we propose an alternative construction based on FS-KEM and AEAD (instead of FS-PKE and collision-resistant hash functions). Without a formal proof, we claim that the security guarantees of both constructions are identical under suitable assumptions. The performance differences depend on the performance of the underlying building blocks. As mentioned above, ciphertexts of the FS-PKE construction in Figure 6 grow based on the FS-PKE's encryption overhead o_{FSE} and the length of the hash l_{H}: $|c| = r * (o_{\mathrm{FSE}} + l_{\mathrm{H}}) + |k|$, where r is the number of re-encryptions. Ciphertexts of the FS-KEM construction in Figure 9 grow based on the AEAD's encryption overhead o_{AE} and the FS-KEM's ciphertext length l_{FSK}: $|c| = r * (l_{\mathrm{FSK}} + o_{\mathrm{AE}}) + |k|$. The decryption time for the FS-PKE based construction is linear in the number of re-encryptions times the FS-PKE decryption time. If FS-KEM decryptions can be parallelized, the decryption time for the FS-KEM based construction is only linear in the number of re-encryptions times the AEAD decryption time. We also note that (without assuming Random Oracles) collision resistant hash functions can only be built from structured algebraic assumptions (see e.g., [BD19]), while AEAD can be built from symmetric primitives.

7 Minimizing Local State Size with External Storage

The lower bound of Theorem 6 shows that the secret state st_{t_0,t_1} of any IKEM(R) scheme must be of size at least $t_1 - t_0$. Storing a secret state of such large size may be cumbersome for the receiver. Therefore, in this section, we introduce the IRAM primitive which the receiver may use to split st_{t_0,t_1} into a small secret component $st^{sec}_{t_0,t_1}$ (of size $O(1)$ in our construction) and public component $st^{pub}_{t_0,t_1}$ (still of size $O(t_1 - t_0)$ in our construction). Only the former needs to be securely stored by the receiver, while the latter can be stored by a server. IRAM allows for a generic interface to perform reads, writes, and deletions on the original state st_{t_0,t_1}, with minimal overhead. Any data that is stored in st_{t_0,t_1} should remain

Initialization: Set $i^* \leftarrow -1$ and $D[\cdot] \leftarrow \perp$. Then initialize $(st_{sec}, st_{pub}) \leftarrow_\$ \text{IRM.init}(1^\lambda)$ and output st_{pub}.

Write$(i, \widetilde{C}, d)$:

```
1. execute $(st_{pub}, C) \leftarrow$
     $\text{IRM.srvr-op}(st_{pub}, i, \text{write})$
2. $C \leftarrow \widetilde{C}$
3. execute $(st_{sec}, C') \leftarrow_\$$
     $\text{IRM.write}(st_{sec}, C, i, d)$
4. execute $st_{pub} \leftarrow$
     $\text{IRM.srvr-up}(st_{pub}, i, C')$
5. if $C' \neq \perp$, set $D[i] \leftarrow d$ and if
     (also) $i = i^*$, set $i^* \leftarrow -1$
6. return $C'$
```

Read$(i, \widetilde{C})$:

```
1. execute $(st_{pub}, C) \leftarrow$
     $\text{IRM.srvr-op}(st_{pub}, i, \text{read})$
2. set att $\leftarrow 0$
3. if $C \neq \widetilde{C}$, set $C \leftarrow \widetilde{C}$ and att $\leftarrow 1$
4. execute $(st_{sec}, d) \leftarrow$
     $\text{IRM.read}(st_{sec}, C, i)$
5. if (att $= 0$ and $d \neq D[i]$) or
     att $= 1$ and $d \notin \{D[i], \perp\}$), **win**
```

Expose():

```
1. return $\perp$ if $i^* \neq -1$
2. return $st_{sec}$
```

Chall$(i, \widetilde{C}, d_0, d_1)$:

```
1. execute $(st_{pub}, C) \leftarrow$
     $\text{IRM.srvr-op}(st_{pub}, i, \text{write})$
2. $C \leftarrow \widetilde{C}$
3. flip random coin $b \leftarrow_\$ \{0, 1\}$
4. execute $(st_{sec}, C') \leftarrow_\$$
     $\text{IRM.write}(st_{sec}, C, i, d_b)$
5. execute $st_{pub} \leftarrow$
     $\text{IRM.srvr-up}(st_{pub}, i, C')$
6. if $C' \neq \perp$, set $D[i] \leftarrow d_b$ and $i^* \leftarrow i$
7. disable **Chall** and return $C'$
```

Del$(i, \widetilde{C})$:

```
1. execute $(st_{pub}, C) \leftarrow$
     $\text{IRM.srvr-op}(st_{pub}, i, \text{del})$
2. $C \leftarrow \widetilde{C}$
3. execute $(st_{sec}, C') \leftarrow_\$$
     $\text{IRM.del}(st_{sec}, C, i)$
4. execute $st_{pub} \leftarrow$
     $\text{IRM.srvr-up}(st_{pub}, i, C')$
5. if $C' \neq \perp$, set $D[i] \leftarrow \perp$ and (also)
     if $i = i^*$ set $i^* \leftarrow -1$
6. return $C'$
```

Fig. 7. IRAM correctness and security game. Text written in green is only needed for RIND_{IRM} security.

secure even if the adversary obtains several of the secret states before the data is written to st_{t_0,t_1} and after it is removed from st_{t_0,t_1}. We provide a definition that allows the server holding $st^{pub}_{t_0,t_1}$ to be actively corrupted (i.e., to deviate from the protocol arbitrarily), and also a definition in which the server is honest-but-curious). We call an IRAM that is secure even against an actively corrupted server, *robust*. The IRAM primitive can easily be composed with IKEM(R) to yield a secure IKEM(R) scheme with small secret state.

Interval RAM Definition The IRAM primitive splits the storage of a dynamically-sized database into a secret component stored by the client and a public component stored by the server. For each read, write, and deletion operation on some virtual location i of the database, the server first executes a corresponding algorithm on the public state which outputs those cells C of the new public state that are relevant for the client-side operation. The server algorithm is deterministic based on the virtual location i of the database on which the client is operating.[7] The client then uses C received from the server to perform the operation and possibly uploads new cells C' with which the server should update the public state. In the case of an actively corrupted server, the adversary may send incorrect cells $\widetilde{C}$ to the client for operations, while for honest-but-curious server, the cells sent by the server are always the correct ones, C.

Syntax An *Interval RAM* (IRAM) scheme IRM is a tuple of algorithms IRM = (IRM.init, IRM.srvr-op, IRM.read, IRM.write, IRM.del, IRM.srvr-up) with the following syntax:

– $\mathrm{IRM.init}(1^\lambda) \rightarrow_\$ (st_{sec}, st_{pub})$ initializes the secret and public state of IRAM.
– $\mathrm{IRM.srvr\text{-}op}(st_{pub}, i, op) \rightarrow (st_{pub}, C)$ the server executes operation $op \in \{\mathrm{read}, \mathrm{write}, \mathrm{del}\}$ on virtual cell i of the database, updating st_{pub}, and returns those locations of st_{pub} that are needed by the client to execute op, via C.
– $\mathrm{IRM.read}(st_{sec}, C, i) \rightarrow (st_{sec}, , d)$ using cells C of st_{pub} provided by the server, the client returns the i-th entry of the database, d.
– $\mathrm{IRM.write}(st_{sec}, C, i, d) \rightarrow_\$ (st_{sec}, C')$ using cells C of st_{pub} provided by the server, the client writes data d to the i-th entry of the database. In doing so, the client returns new public cells C' relevant to virtual cell i with which the server will replace the corresponding cells in st_{pub}.
– $\mathrm{IRM.del}(st_{sec}, C, i) \rightarrow_\$ (st_{sec}, C')$ using cells C provided by the server, the client deletes the i-th database entry. In doing so, the client returns new public cells C' relevant to virtual cell i with which the server will replace the corresponding cells in st_{pub}.
– $\mathrm{IRM.srvr\text{-}up}(st_{pub}, C) \rightarrow st_{pub}$ the server places cells C in st_{pub} (the location of these cells in st_{pub} are implicitly encoded in C).

Correctness and Security Now we define the correctness and security for an IRAM scheme. Intuitively, correctness dictates that when reading the i-th virtual location of the database, the data which was last written to the i-th virtual location must be returned. In the case of actively corrupted server, if the adversary does not honestly execute IRM.srvr-op() or IRM.srvr-up at some point, then the scheme can reject by outputting $\perp$. Security dictates that if any data d is written to cell i of the database, then it should remain private even if the adversary obtained several of the IRAM secret states *before* the write operation and obtains several secret states *after* data d of cell i is overwritten or deleted.

More formally, we define the correctness and security games XIND, $\mathrm{X} \in \{\mathrm{R}, \epsilon\}$ for IRAM in Figure 7. The former is for R̲obust security, in which the adversary may arbitrarily deviate from the protocol specification, and the latter

[7] Deterministic server operations are common in outsourced database primitives, see e.g., [BEG+91, DNRV09, BDY21, BDT22].

is for non-robust security, in which the adversary is assumed to be honest-but-curious. The green text in Figure 7 is only for $\mathrm{RIND}_{\mathrm{IRM}}$. The game starts by initializing the IRAM via IRM.init, and outputting the public state st_{pub} to the adversary. It also stores variable $i^* \leftarrow -1$, which indicates if there is an active challenge, and if so the cell for which this challenge has been queried ($i^* = -1$ means there is no active challenge). For each **Write** query, the adversary specifies the virtual cell i and data d to write. The oracle first executes $(st_{pub}, C) \leftarrow$ IRM.srvr-op$(st_{pub}, i, \text{write})$ to perform the server-side write operation and obtain the cells C necessary for the client-side operation; in the case of game $\mathrm{RIND}_{\mathrm{IRM}}$, the adversary actually inputs cells $\widetilde{C}$ to be used for the client-side operation (which is reflected by the oracle setting $C \leftarrow \widetilde{C}$). The oracle then executes IRM.write on input C, i, and d to obtain new secret state st_{sec} and cells C' which it uses to update st_{pub} via IRM.srvr-up. Then, the oracle stores data d in its own dictionary $D[i] \leftarrow d$ and checks if there is an active challenge for cell i ($i^* = i$), and if so resets $i^* \leftarrow -1$, since this query will overwrite the challenge data. This step is only applied in game $\mathrm{RIND}_{\mathrm{IRM}}$ if $C' \neq \perp$; i.e., only if the IRM.write operation was successful. Finally, the oracle returns the updated C' to the adversary. Oracle **Del** is specified in exactly the same way as **Write**, except instead of executing IRM.write, it executes IRM.del on virtual cell i.

For each query to **Chall**, the game acts similarly as to **Write**, except it sets challenge cell $i^* \leftarrow i$ (if $C' \neq \perp$ in the case of $\mathrm{RIND}_{\mathrm{IRM}}$; i.e., the IRM.write operation was successful), and flips a random coin b to determine on which of d_0 or d_1 IRM.write should be executed. Then, the game disables the oracle. Oracle **Read** takes as input from the adversary virtual cell i to read. The oracle first executes $(st_{pub}, C) \leftarrow$ IRM.srvr-op$(st_{pub}, i, \text{read})$ to perform the server-side read operation and obtain the cells C necessary for the client-side operation; as with the other oracles, in the case of game $\mathrm{RIND}_{\mathrm{IRM}}$, the adversary actually inputs cells $\widetilde{C}$ to be used for the client-side operation. Also in the case of game $\mathrm{RIND}_{\mathrm{IRM}}$, if these adversarial cells do not match the cells from the honest IRM.srvr-op operation, $\widetilde{C} \neq C$, then the oracle sets att $\leftarrow 1$ to denote an active attack by the adversary. Note that the deterministic nature of IRM.srvr-op, IRM.srvr-up enables the game to detect active attacks in this way. Next, the oracle executes IRM.read on input i and cells C to receive data d. In the case of $\mathrm{RIND}_{\mathrm{IRM}}$, if att $= 0$, the oracle checks that d is equal to $D[i]$ which was last successfully written to cell i of the database; if att $= 1$, the oracle checks that either d is still equal to $D[i]$ or $d = \perp$ (indicating detection of the adversarial attack). For whichever value of att, if the check fails, the adversary wins the game, denoted by keyword **win** (in this case, the game just outputs the challenge bit b). In the non-robust game $\mathrm{IND}_{\mathrm{IRM}}$, the oracle only checks that $d = D[i]$. Finally, **Expose** simply returns the current secret state st_{sec} to the adversary, only if $i^* = -1$, i.e., only if there is not an active challenge.

Definition 16. *For* $\mathrm{X} \in \{\epsilon, \mathrm{R}\}$,[8] *an IRAM scheme* IRM *is* $(T, \varepsilon^{\mathrm{xind}}_{\mathrm{IRM}})$-secure *if for all adversaries* $\mathcal{A}$ *playing the security game* $\mathrm{XIND}_{\mathrm{IRM}}$ *and running in time* T*:* $\Pr[b \leftarrow_\$ \mathrm{XIND}_{\mathrm{IRM}}(\mathcal{A})] \leq 1/2 + \varepsilon^{\mathrm{xind}}_{\mathrm{IRM}}$.

[8] ϵ indicates the ‘empty string’.

7.1 Interval RAM Construction

In this section, we present our IRM construction. It is based on the FS eRAM of [BDY21, BDT22] and FS Memory Checker of [BDY21]. For n stored entries, our construction has secret state size of $O(1)$ and read/write overhead of $O(\log n)$, which matches the lower bound of [BDY21] for FS encrypted RAM. The construction utilizes a generic tree data structure, which we describe first below. As shown in [BDT22] (c.f. Appendix A), such a tree data structure can be efficiently instantiated using 2-3 trees or left-leaning red-black trees.

Tree Data Structure In this paragraph, we give a general description of a tree data structure, τ. We denote by $\tau.r$ the root of tree τ. Every node v in τ has associated data $v.x$ and unique identifier ℓ_v. For each node v in τ, we denote by $deg(v)$ its number of children. The maximum degree of any node v of τ is denoted as $deg(\tau)$. We call the children of a node v, $v.c_j$, for $j \in [deg(v)]$, and its parent $v.p$. We say that a node w is an *ancestor* of some node v if w is on the path from v to $\tau.r$. A node v is a leaf if $\deg(v) = 0$. Let the number of leafs in τ be n. For each leaf $i \in [0, n-1]$ we denote by $\tau.path(i)$ the sequence of data $(x_1, \ldots, x_{d(i)})$ associated with the nodes on the path from $\tau.r$ to leaf i, excluding $\tau.r$; the depth $d(i)$ of leaf i is the length of $\tau.path(i)$. The height $h(\tau)$ of τ is equal to the maximum depth of its leaves, i.e., $\max_{i\in[0,n-1]} d(i)$. For most commonly used efficient tree data structures, $h(\tau) = O(\log n)$. This is indeed true for 2-3 trees and left-leaning red-black trees (see [BDT22]).

Let the co-path to some leaf i be the siblings of nodes on the path from $\tau.r$ to leaf i. For $i \in [0, n-1]$, we denote by $\tau.copath(i)$ the sequence of data $(x_{2,1}, \ldots, x_{2,\deg(\tau.r)}, \ldots, x_{d(i),1}, \ldots, x_{d(i),\deg(i.p)})$ associated with the nodes on the co-path to leaf i. We refer to a connected subtree of the tree that includes the root as a skeleton *skel* of the tree.

Leaves of τ can be explicitly added or removed via the following algorithms. A new leaf v can be added to tree τ via $\tau.\text{add}(v)$. Similarly, a leaf v can be removed from tree τ via $\tau.\text{remove}(v)$. These operations return two skeletons: $(skel, skel')$. The former, *skel*, consists of any new nodes added to τ during the operation, as well as any nodes in the original tree τ whose subtree was modified (which by definition, includes the root $\tau.r$). The latter, $skel'$, consists of those nodes in the original tree τ who share children with any nodes of *skel*, as well as all of their ancestors, up to $\tau.r$. For most commonly used efficient tree data structures, the resulting skeletons $(skel, skel')$ will consist of nodes only on the path $\tau.path(v)$ and/or co-path $\tau.copath(v)$ of the added/removed leaf v. Indeed, for 2-3 trees and left-leaning red-black trees, the size of $(skel, skel')$ for add and remove operations is $O(\log n)$ (see [BDT22]).

Construction We present our construction for IRM in Figure 8. Our construction makes use of a symmetric-key encryption scheme, SK. For security in the robustness game, RIND_{IRM}, we also use a UOWHF, $\mathcal{H}$. We explain here our construction secure in RIND_{IRM}; the construction secure in IND_{IRM} is obtained by simply removing the hashing. At a high level, our construction is based on a tree

τ, where each node v in the tree, besides the leaves, has some SK key k_v associated with it. The n leaves have the n current data elements in the database associated with them. The key k_v (resp. data d_v) at each node v (resp. leaf), besides the root $\tau.r$, is encrypted by the key at its parent $k_{v.p}$; $c_v \leftarrow_\$ \text{SK.enc}(k_{v.p}, k_v)$ (resp. $c_v \leftarrow_\$ \text{SK.enc}(k_{v.p}, d_v)$). Each node v also contains a hash function h_v and the hash y_v under h_v of the concatenated associated data of its children. These ciphertexts, hash functions, and hashes are the actual associated data $v.d \leftarrow c_v$ of each node v of τ (along with some unique identifier), except the root. Then, we have that $st_{sec} = k_{\tau.r}, h_{\tau.r}, y_{\tau.r}$, where $y_{\tau.r}$ is the hash of the children of the root, under $h_\tau.r$, and $st_{pub} = \tau$ (but without the root).

Based on this, reading entry i from the database is simple. The client fetches from the server the ciphertexts and hashes along the path and copath from $\tau.r$ to leaf i. Starting from the root, down to the leaf, the client checks at each node v along the path that the hash value y_v matches the hash of the associated data at the children of v, under function h_v, and aborts if not. Then, the client uses k_v to decrypt the ciphertext $c_{v.c^*}$ at the child of v on the path to leaf i, $v.c^*$, to obtain $k_{v.c^*}$. It continues this until it decrypts d_i from the ciphertext stored at the leaf i.

For writing data d to entry i of the database in IRM.write, there is first a server-side operation on τ, specified by IRM.srvr-op(τ, i, write). In this algorithm, it is first checked whether the leaf with unique identifier i already exists in the tree τ. If so, the server sets *skel* and *skel′* to be $\tau.path(i)$. Otherwise, the server runs $(skel, skel') \leftarrow \tau.\text{add}(i)$. The client-side operation then uses $(skel, skel')$ sent by the server to perform the rest of the write. Indeed, the client runs subroutine skel-mod on input $(skel, skel', i, d)$. This subroutine first initializes dictionary D and sets $D[i] \leftarrow d$. Next, similarly to IRM.read described above, starting at the root it recursively checks the hashes at the nodes v in *skel′* and then decrypts the ciphertexts at these nodes, along with the children v of *skel′* that are not in *skel′*, and sets $D[v]$ to be the corresponding decrypted key/data. This is done to retain knowledge of the keys/data at other nodes in the tree that are used to obtain the data at other leaves in the tree (virtual cells of the IRAM). Next, recursively for the nodes v in *skel* besides leaf i, starting at those nodes who are parents of nodes that are either leaves or are not in *skel*, up to the root, it samples new SK keys k_v, which it uses to encrypt the data of the children of v (obtained via $D[v]$). Observe that if the child of v is also in *skel*, then the algorithm has the data of the child node (either a freshly sampled SK key, or $D[i] = d$ if the child is leaf i); otherwise, if the child $v.c$ of v is not in *skel*, then by the definition of *skel′*, it must also be some child of a node v' in *skel′*, and so in the recursive decryption above, the algorithm retrieved its data and stored it in $D[v.c]$. The obtained ciphertexts are then used to replace what was previously stored at those children in *skel*. It also samples new hash function h_v and uses it to compute the hash y_v of the associated data of the children of v, then replaces what was previously stored at v in *skel* with this hash and hash function. The skel-mod subroutine finally returns root data $(k_{\tau.r}, h_{\tau.r}, y_{\tau.r})$ and

```
IRM.init($1^\lambda$):
  1. sample $k_{\tau.r} \leftarrow_\$ \mathrm{SK.gen}(1^\lambda)$
  2. set $h_{\tau.r}, y_{\tau.r} \leftarrow \perp$
  3. return $((k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), \perp)$

IRM.srvr-op($\tau, i, op$):
  1. if $op$ = read: return
       $(\tau, (\tau.path(i), \tau.copath(i))$
  2. if $op$ = write:
     (a) if leaf $i \in \tau$: return
           $(\tau, (\tau.path(i), \tau.path(i))$
     (b) else: return $(\tau, \tau.\mathrm{add}(i))$
  3. else: return $(\tau, \tau.\mathrm{remove}(i))$

IRM.write($(k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), C, i, d$):
  1. set $(skel, skel') \leftarrow C$
  2. run $((k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), skel) \leftarrow$
       skel-mod$(skel, skel', i, d)$
  3. return $((k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), skel)$

IRM.read($(k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), C, i$):
  1. let $(\tau.path(i), \tau.copath(i)) \leftarrow C$
  2. let $(x_1, \ldots, x_{d(i)}) \leftarrow \tau.path(i)$
  3. let $(x_{2,1}, \ldots, x_{d(i),\deg(d(i))}) \leftarrow$
       $\tau.copath(i)$
  4. parse each $x_i$ and $x_{i,j}$ as $(c, h, y)$
  5. if $y_{\tau.r} \neq h_1(x_{1,1}, \ldots, x_{1,\deg(\tau.r)})$:
       return $\perp$
  6. let $k_0 = k_{\tau.r}$
  7. For $j \in [d(i) - 1]$:
     (a) if $y_j \neq$
           $h_j(x_{j+1,1}, \ldots, x_{j+1,\deg(j)})$ :
           return $\perp$
     (b) decrypt $k_j \leftarrow \mathrm{SK.dec}(k_{j-1}, c_j)$
  8. decrypt $d \leftarrow \mathrm{SK.dec}(k_{d(i)-1}, c_{d(i)})$
  9. return $((k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), d)$

skel-mod($skel, skel', i, d$):
  1. set $D[\cdot] \leftarrow \perp, D[i] \leftarrow d$
  2. for each $v \in skel \cup skel'$:
       parse $(c_v, h_v, y_v) \leftarrow v.x$
  3. starting with $v \leftarrow \tau.r \in skel'$:
     (a) if $y_v \neq h_v(x_{v.c_1}, \ldots, x_{v.c_{\deg(v)}})$:
           return $\perp$
     (b) for each $j \in [deg(v)]$:
        i. retrieve $c_{v.c_j}$ from $skel'$
        ii. decrypt $D[v.c_j] \leftarrow$
              $\mathrm{SK.dec}(k_v, c_{v.c_j})$
        iii. if $v.c_j \in skel'$ and $\deg(v.c_j) > 0$:
              set $k_{v.c_j} \leftarrow D[v.c_j]$ and recurse
                on $v.c_j$
  4. for each $v \in skel$ s.t. $\deg(v) \neq 0$ and
       $\forall j \in \deg(v), \deg(v.c_j) = 0$ or
       $v.c_j \notin skel$:
     (a) sample $k_v \leftarrow_\$ \mathrm{SK.gen}(1^\lambda)$
     (b) sample $h_v \leftarrow_\$ \mathcal{H}_\lambda$
     (c) set $D[v] \leftarrow k_v$
     (d) for each $j \in [deg(v)]$:
        i. set $m \leftarrow D[v.c_j]$
        ii. encrypt $c_{v.c_j} \leftarrow_\$ \mathrm{SK.enc}(k_v, m)$
        iii. set $c_{v.c_j} \leftarrow c_{v.c_j}$ in $skel$
     (e) set $(h_v, y_v) \leftarrow (h_v,$
           $h_v(x_{v.c_1}, \ldots, x_{v.c_{\deg(v)}}))$ in $skel$
  5. recurse bottom-up on the parents of
       the nodes just processed until $\tau.r$
  6. return $(k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), skel)$

IRM.del($(k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), C, i$):
  1. set $(skel, skel') \leftarrow C$
  2. run $((k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), skel) \leftarrow$
       skel-mod$(skel, skel', \perp, \perp)$
  3. return $((k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), skel)$

IRM.srvr-up($\tau, skel$):
  1. replace the corresponding nodes in $\tau$
       with $skel$
  2. return $\tau$
```

Fig. 8. IRAM construction. The text written in green is only needed for the construction secure in $\mathrm{RIND}_{\mathrm{IRM}}$.

skel to IRM.write. IRM.srvr-up(τ, *skel*) can then replace the relevant associated data of nodes from this operation in τ.

For deleting entry i from the database, the server first runs $(\mathit{skel}, \mathit{skel}') \leftarrow \tau.\text{remove}(i)$ in IRM.srvr-op(τ, i, del). Then similarly to IRM.write above, the client runs skel-mod on input $(\mathit{skel}, \mathit{skel}', \perp, \perp)$, and receives fresh root data $(k_{\tau.r}, h_{\tau.r}, y_{\tau}.r)$ and *skel*, the latter of which IRM.srvr-up uses to replace the relevant associated data of nodes in τ.

Correctness and Security We now show that our IRM construction of Figure 8 is correct and secure. It is easy to see why IRM.read$((k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), C_r, i)$ returns the proper data d immediately after IRM.write$((k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), C_w, i, d)$ when C_r, C_w are honestly computed by the server—leaf i is in the skeleton *skel* of the latter operation and thus a path of encryptions and hashes from the root key to the leaf will exist in τ. It is also easy to see that correctness of such a write to cell i holds even after several executions of IRM.write$((k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), C, j, d)$ and IRM.del$((k_{\tau.r}, h_{\tau.r}, y_{\tau.r}), C, j, d)$, for $j \neq i$, as long as if the cells C are always honestly computed by the server. This is because both algorithms re-encrypt/re-hash the keys/data of those children of nodes in *skel* that are not in *skel* since by definition, these children must also be children of some nodes in *skel*$'$. Furthermore, these children must in fact be ancestors of the remaining leaves $l \neq j \in \tau$, including leaf i, since by definition, each l will never be in *skel*. Thus, there will always be a path of correct encryptions and hashes from the root key to leaf i.

Security follows first from the fact that by UOWHF security, the adversary will never be able to provide incorrect cells C to the client. Indeed, starting at the root, for any node v in the tree, finding any associated data of the children (which includes the hash at the children) that hashes to the same value as that which is stored at v would break the UOWHF. Furthermore, regarding privacy, for any execution of IRM.write or IRM.del, any ancestors of leaf i in τ before the operation that remain in the tree after the operation (including the root) must by definition be in *skel*. Thus their associated key must be re-sampled. Therefore, even given the root key $k_{\tau.r}$ afterwards, the old data d' stored at leaf i is secure. Conversely, any old root key exposed by the adversary before a IRM.write operation will not give any information on the new data d' stored at leaf i, since each ancestor of leaf i receives a fresh key during the operation. We prove the following Theorem in Appendix D.

Theorem 5. *Let* SK *be a* $(T, \varepsilon_{\text{SK}})$*-secure symmetric-key encryption scheme and* $\mathcal{H}$ *be a* $(T, \varepsilon_{\mathcal{H}})$*-secure universal one-way hash function family. Then the* IRM *construction of Figure 8 instantiated with tree* τ *is correct,* $(T', h(\tau) \cdot \varepsilon_{\text{SK}})$*-secure in the* IND_{IRM} *game of Figure 7, and* $(T', h(\tau) \cdot (\varepsilon_{\text{SK}} + T' \cdot \varepsilon_{\mathcal{H}}))$*-secure in the* RIND_{IRM} *game of Figure 7, for* $T' \approx T$*.*

Corollary 3. *Let* SK *be a* $(T, \varepsilon_{\text{SK}})$*-secure symmetric-key encryption scheme and* $\mathcal{H}$ *be a* $(T, \varepsilon_{\mathcal{H}})$*-secure universal one-way hash function family. Then the* IRM *construction of Figure 8 instantiated with either a 2-3 tree or a left-leaning red-black tree* τ *is correct,* $(T', O(\log N) \cdot \varepsilon_{\text{SK}})$*-secure in the* IND_{IRM} *game of Figure 7,* *and* $(T', O(\log N) \cdot (\varepsilon_{\mathrm{SK}} + T' \cdot \varepsilon_{\mathcal{H}}))$*-secure in the* $\mathrm{RIND}_{\mathrm{IRM}}$ *game of Figure 7, for* $T' \approx T$ *and* N *the maximum number of entries ever in the database.*

Remark 1. As noted in [BDY21], in practice, a single CRHF can be used in place of a UOWHF family so that there is no need to regenerate the hash functions at the nodes of *skel* for each IRM.write and IRM.del operation, nor include them in the hashes at each node. We use a UOWHF family since it is possible to construct one given any OWF.

Acknowledgments

This paper was prepared in part for information purposes by the Artificial Intelligence Research group of JPMorgan Chase & Co and its affiliates ("JP Morgan"), and is not a product of the Research Department of JP Morgan. JP Morgan makes no representation and warranty whatsoever and disclaims all liability, for the completeness, accuracy or reliability of the information contained herein. This document is not intended as investment research or investment advice, or a recommendation, offer or solicitation for the purchase or sale of any security, financial instrument, financial product or service, or to be used in any way for evaluating the merits of participating in any transaction, and shall not constitute a solicitation under any jurisdiction or to any person, if such solicitation under such jurisdiction or to such person would be unlawful. 2024 JP Morgan Chase & Co. All rights reserved.

References

AAB+21. Joël Alwen, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Grafting key trees: Efficient key management for overlapping groups. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part III*, volume 13044 of *LNCS*, pages 222–253. Springer, Heidelberg, November 2021.

ACDT20. Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Heidelberg, August 2020.

ACJM20. Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 261–290. Springer, Heidelberg, November 2020.

AKPS19. Benedikt Auerbach, Eike Kiltz, Bertram Poettering, and Stefan Schoenen. Lossy trapdoor permutations with improved lossiness. In Mitsuru Matsui, editor, *CT-RSA 2019*, volume 11405 of *LNCS*, pages 230–250. Springer, Heidelberg, March 2019.

BBR+23. Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

BBS98. Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 127–144. Springer, Heidelberg, May / June 1998.

BD19. Nir Bitansky and Akshay Degwekar. On the complexity of collision resistant hash functions: New and old black-box separations. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 422–450. Springer, Heidelberg, December 2019.

BDR20. Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 198–228. Springer, Heidelberg, November 2020.

BDRW24a. Alexander Bienstock, Yevgeniy Dodis, Paul Rösler, and Daniel Wichs. Interval key-encapsulation mechanism. In *Advances in Cryptology - ASIACRYPT 2024 - 30th International Conference on the Theory and Application of Cryptology and Information Security, 2024, Proceedings*, Lecture Notes in Computer Science, 2024.

BDRW24b. Alexander Bienstock, Yevgeniy Dodis, Paul Rösler, and Daniel Wichs. Interval key-encapsulation mechanism. *IACR Cryptol. ePrint Arch.*, 2024. Full version of this article.

BDT22. Alexander Bienstock, Yevgeniy Dodis, and Yi Tang. Multicast key agreement, revisited. In Steven D. Galbraith, editor, *CT-RSA 2022*, volume 13161 of *LNCS*, pages 1–25. Springer, Heidelberg, March 2022.

BDY21. Alexander Bienstock, Yevgeniy Dodis, and Kevin Yeo. Forward secret encrypted RAM: Lower bounds and applications. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part III*, volume 13044 of *LNCS*, pages 62–93. Springer, Heidelberg, November 2021.

BEG$^+$91. Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *32nd FOCS*, pages 90–99. IEEE Computer Society Press, October 1991.

BL96. Dan Boneh and Richard J Lipton. A revocable backup system. In *USENIX Security Symposium*, pages 91–96, 1996.

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

BRT23. Alexander Bienstock, Paul Rösler, and Yi Tang. Asmesh: Anonymous and secure messaging in mesh networks using stronger, anonymous double ratchet. In *CCS '23: 2023 ACM SIGSAC Conference on Computer and Communications Security 2023.* ACM, 2023.

BRV20. Fatih Balli, Paul Rösler, and Serge Vaudenay. Determining the core primitive for optimally secure ratcheting. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 621–650. Springer, Heidelberg, December 2020.

CDV21. Gwangbae Choi, F. Betül Durak, and Serge Vaudenay. Post-Compromise Security in Self-Encryption. In Stefano Tessaro, editor, *2nd Conference on Information-Theoretic Cryptography (ITC 2021)*, volume 199 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:23, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

CH07. Ran Canetti and Susan Hohenberger. Chosen-ciphertext secure proxy re-encryption. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 185–194. ACM Press, October 2007.

CHK03. Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 255–271. Springer, Heidelberg, May 2003.

CKN03. Ran Canetti, Hugo Krawczyk, and Jesper Buus Nielsen. Relaxing chosen-ciphertext security. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 565–582. Springer, Heidelberg, August 2003.

DDLM19. Alex Davidson, Amit Deo, Ela Lee, and Keith Martin. Strong post-compromise secure proxy re-encryption. In Julian Jang-Jaccard and Fuchun Guo, editors, *ACISP 19*, volume 11547 of *LNCS*, pages 58–77. Springer, Heidelberg, July 2019.

DKW21. Yevgeniy Dodis, Harish Karthikeyan, and Daniel Wichs. Updatable public key encryption in the standard model. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part III*, volume 13044 of *LNCS*, pages 254–285. Springer, Heidelberg, November 2021.

DNRV09. Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. How efficient can memory checking be? In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 503–520. Springer, Heidelberg, March 2009.

DRS04. Yevgeniy Dodis, Leonid Reyzin, and Adam Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 523–540. Springer, Heidelberg, May 2004.

DY91. Alfredo De Santis and Moti Yung. On the design of provably secure cryptographic hash functions. In Ivan Damgård, editor, *EUROCRYPT'90*, volume 473 of *LNCS*, pages 412–431. Springer, Heidelberg, May 1991.

FKKP19. Georg Fuchsbauer, Chethan Kamath, Karen Klein, and Krzysztof Pietrzak. Adaptively secure proxy re-encryption. In Dongdai Lin and Kazue Sako, editors, *PKC 2019, Part II*, volume 11443 of *LNCS*, pages 317–346. Springer, Heidelberg, April 2019.

GS02. Craig Gentry and Alice Silverberg. Hierarchical ID-based cryptography. In Yuliang Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 548–566. Springer, Heidelberg, December 2002.

HPS23. Calvin Abou Haidar, Alain Passelègue, and Damien Stehlé. Efficient updatable public-key encryption from lattices. In *Advances in Cryptology - ASIACRYPT 2023 - 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4-8, 2023, Proceedings, Part V*, volume 14442 of *Lecture Notes in Computer Science*, pages 342–373. Springer, 2023.

HWZ07. Qiong Huang, Duncan S. Wong, and Yiming Zhao. Generic transformation to strongly unforgeable signatures. In Jonathan Katz and Moti Yung, editors, *ACNS 07*, volume 4521 of *LNCS*, pages 1–17. Springer, Heidelberg, June 2007.

JMM19. Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 159–188. Springer, Heidelberg, May 2019.

JS18. Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018.

LV08. Benoît Libert and Damien Vergnaud. Unidirectional chosen-ciphertext secure proxy re-encryption. In Ronald Cramer, editor, *PKC 2008*, volume 4939 of *LNCS*, pages 360–379. Springer, Heidelberg, March 2008.

MPW23. Peihan Miao, Sikhar Patranabis, and Gaven J. Watson. Unidirectional updatable encryption and proxy re-encryption from DDH. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part II*, volume 13941 of *LNCS*, pages 368–398. Springer, Heidelberg, May 2023.

NY89. Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *21st ACM STOC*, pages 33–43. ACM Press, May 1989.

Ode09. Goldreich Oded. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, USA, 1st edition, 2009.

PM16. Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm, 2016. https://signal.org/docs/specifications/doubleratchet/.

PR18a. Bertram Poettering and Paul Rösler. Asynchronous ratcheted key exchange. Cryptology ePrint Archive, Report 2018/296, 2018. https://eprint.iacr.org/2018/296.

PR18b. Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 3–32. Springer, Heidelberg, August 2018.

PW08. Chris Peikert and Brent Waters. Lossy trapdoor functions and their applications. In Richard E. Ladner and Cynthia Dwork, editors, *40th ACM STOC*, pages 187–196. ACM Press, May 2008.

RMS18. Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in signal, whatsapp, and threema. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 415–429. IEEE, 2018.

RSS23. Paul Rösler, Daniel Slamanig, and Christoph Striecks. Unique-path identity based encryption with applications to strongly secure messaging. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 3–34. Springer, Heidelberg, April 2023.

SC09. Jun Shao and Zhenfu Cao. CCA-secure proxy re-encryption without pairings. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 357–376. Springer, Heidelberg, March 2009.

Sha02. Ronen Shaltiel. Recent developments in explicit constructions of extractors. *Bulletin of the EATCS*, 77:67–95, 01 2002.

A Additional Preliminaries

Universal One-Way Hash Function A *Universal One-Way Hash Function* (UOWHF) family $\mathcal{H}_\lambda = \{h : \mathcal{X}_\lambda \to \mathcal{Y}_\lambda\}_\lambda$ is a family of compressing functions such that for every $x \in \mathcal{X}_\lambda$, given random $h \in \mathcal{H}_\lambda$, it is hard to find $y \neq x$ such that $h(x) = h(y)$.

Definition 17. *A family of functions* $\mathcal{H}_\lambda = \{h : \mathcal{X}_\lambda \to \mathcal{Y}_\lambda\}_\lambda$ *is* $(T, \varepsilon_{\mathcal{H}})$-secure *if for every* λ*,* $|\mathcal{Y}_\lambda| < |\mathcal{X}_\lambda|$*, and for all adversaries running in time* T

$$\Pr[y \leftarrow_\$ \mathcal{A}(h, x) : x \leftarrow_\$ \mathcal{A}(1^\lambda); h \leftarrow_\$ \mathcal{H}_\lambda; x \neq y; h(x) = h(y)] \leq \varepsilon_{\mathcal{H}}.$$

UOWHFs can be built from OWFs [DY91, NY89].

Pseudorandom Generators A *Pseudorandom Generator* (PRG) is a function $\mathrm{G} : \mathcal{S} \to \mathcal{R}$ such that for random seed $s \leftarrow_\$ \mathcal{S}$, $\mathrm{G}(s)$ is indistinguishable from random $r \leftarrow \mathcal{R}$.

Definition 18. *PRG* G *is* $(T, \varepsilon_{\mathrm{G}})$-secure *if for all adversaries* $\mathcal{A}$ *running in time* T*:*

$$\Pr[b \leftarrow_\$ \mathcal{A}(r_b) : b \leftarrow_\$ \{0, 1\}; s \leftarrow_\$ \mathcal{S}; r_0 \leftarrow \mathrm{G}(s); r_1 \leftarrow_\$ \mathcal{R}] \leq 1/2 + \varepsilon_{\mathrm{G}}.$$

Symmetric-Key Encryption

Syntax A *Symmetric-Key Encryption* (SKE) scheme SK is a tuple of algorithms $\mathrm{SK} = (\mathrm{SK.gen}, \mathrm{SK.enc}, \mathrm{SK.dec})$ with the following syntax:

– $\mathrm{SK.gen}(1^\lambda) \to_\$ k$ generates random key k.
– $\mathrm{SK.enc}(k, m) \to_\$ c$ encrypts message m to key k and outputs ciphertex t c.
– $\mathrm{SK.dec}(c, k) \to c$ decrypts ciphertext c to message m using key k.

Correctness

Definition 19. *A SKE scheme* SK *is* correct *if for all* $\mathrm{SK.gen}(1^\lambda) \to_\$ k$*,* $m \in \mathcal{M}$*, and* $\mathrm{SK.enc}(k, m) \to_\$ c$*,* $\mathrm{SK.dec}(k, c) = m$*.*

Security

Definition 20. *Scheme* SK *is* $(T, \varepsilon_{\mathrm{SK}})$-secure *if for all adversaries* $\mathcal{A}$ *running in time* T*:*

$$\Pr[b \leftarrow_\$ \mathcal{A}^{\mathrm{SK.enc}(k,\cdot)}(c_b) : b \leftarrow_\$ \{0, 1\}; k \leftarrow_\$ \mathrm{SK.gen}(1^\lambda);$$
$$(m_0, m_1) \leftarrow_\$ \mathcal{A}^{\mathrm{SK.enc}(k,\cdot)}; c_b \leftarrow_\$ \mathrm{SK.enc}(k, m_b)] \leq 1/2 + \varepsilon_{\mathrm{SK}},$$

where $\mathcal{A}^{\mathrm{SK.enc}(k,\cdot)}$ *denotes that* $\mathcal{A}$ *gets oracle access to* $\mathrm{SK.enc}(k, \cdot)$*.*

Strongly Unforgeable One-Time Signature Scheme Now we define strongly unforgeable one-time signature (OTS) schemes, which allows a signer to sign a *single* message using its secret key, which can be verified by anyone using the corresponding verification key.

Syntax A *strongly unforgeable one-time signature scheme* (OTS) scheme OTS is a tuple of algorithms OTS $=$ (OTS.gen, OTS.sign, OTS.ver) with the follwing syntax:

– OTS.gen$(1^\lambda) \to_\$ (sk, vk)$ generates a signing key and verification key.
– OTS.sign$(sk, m) \to_\$ \sigma$ signs message m using the signing key and outputs signature σ.
– OTS.ver$(vk, m, \sigma) \to b$ verifies the signature σ on message m using verification key vk and outputs bit $b \in \{0, 1\}$ indicating success or failure.

Correctness

Definition 21. *A OTS scheme* OTS *is* correct *if for all* OTS.gen$(1^\lambda) \to_\$ (sk, vk)$ *and all* m*,* OTS.ver$(vk, m,$ OTS.sign$(sk, m)) = 1$*.*

Security Security of a OTS scheme requires that if an adversary sees a *single* signature σ on a message m of its choice, then the adversary cannot output a different pair $(m', \sigma') \neq (m, \sigma)$ that verifies correctly.

Definition 22. *Scheme* OTS *is* $(T, \varepsilon_{\mathrm{OTS}})$-secure *if for all adversaries* $\mathcal{A}$ *running in time* T*:*

$$\begin{aligned}\Pr[1 = \mathrm{OTS.ver}(vk,\ m', \sigma') : {} & (sk, vk) \leftarrow_\$ \mathrm{OTS.gen}(1^\lambda); m \leftarrow_\$ \mathcal{A}(vk);\\ & \sigma \leftarrow_\$ \mathrm{OTS.sign}(sk, m)\\ & (m', \sigma') \leftarrow_\$ \mathcal{A}(vk, m, \sigma); (m', \sigma') \neq (m, \sigma)] \leq \varepsilon_{\mathrm{OTS}}.\end{aligned}$$

Such OTS schemes can be constructed from one-way functions [Ode09] and more efficiently from collision-resistant hash functions [HWZ07] (which in turn can be built from lossy trapdoor functions [PW08]).

Key-Encapsulation Mechanism Now we define the Key-Encapsulation Mechanism (KEM) primitive, which allows a receiver to generate a key-pair, to which senders can send random keys.

Syntax A *Key-Encapsulation Mechanism* (KEM) scheme KM is a tuple of algorithms KM $=$ (KM.gen, KM.enc, KM.dec) with the following syntax:

– KM.gen$(1^\lambda) \to_\$ (sk, pk)$ generates a key-pair.
– KM.enc$(pk) \to_\$ (k, c)$ takes as input a public key pk and outputs a ciphertext c and the encapsulated key k.
– KM.dec$(sk, c) \to k$ takes in a secret key sk and ciphertext c and outputs the decapsulated key k.

Correctness

Definition 23. *A KEM scheme* KM *is* correct *if for all* KM.gen$(1^\lambda) \to_\$ (sk, pk)$ *and* KM.*enc*$(pk) \to_\$ (k, c)$*,* KM.*dec*$(sk, c) = k$*.*

Security A KEM scheme KM is secure if the key encapsulated to a public key is indistinguishable from random to an attacker without the secret key.

Definition 24. *Scheme* KM *is* $(T, \varepsilon^{\mathrm{ind}}_{\mathrm{KM}}\text{-cpa})$-secure *in the* IND-CPA *security game if for all adversaries $\mathcal{A}$ running in time T:*

$$\Pr[b \leftarrow_{\$} \mathcal{A}(pk, k_b, c^*) : b \leftarrow_{\$} \{0,1\};\ (sk, pk) \leftarrow_{\$} \mathrm{KM.gen}(1^\lambda);\\ (k_0, c^*) \leftarrow_{\$} \mathrm{KM.enc}(pk); k_1 \leftarrow_{\$} \mathcal{K}] \leq 1/2 + \varepsilon^{\mathrm{ind}}_{\mathrm{KM}}\text{-cpa}.$$

IND-CCA security is defined in the same way, except the adversary also gets access to a decryption oracle $\mathbf{Dec}(c)$ that outputs $\mathrm{KM.dec}(sk, c)$ if $c \neq c^*$.

Randomness Extraction We now define some concepts related to randomness extraction that we will need for our IKEM construction. First, we define pairwise-independent hash families.

Definition 25. *A family of functions $\mathcal{H} = \{h : \mathcal{X} \to \mathcal{Y}\}$ is* pairwise-independent *if, for every $x \neq x' \in \mathcal{X}$ and $y, y' \in \mathcal{Y}$,*

$$\Pr[h(x) = y \text{ and } h(x') = y' : h \leftarrow_{\$} \mathcal{H}] = 1/|\mathcal{Y}|^2.$$

Now, we define some information-theoretic concepts. We start with the *statistical distance* between two random variables.

Definition 26. *The* statistical distance *between two random variables X and Y having the same (countable) domain $\mathcal{X}$ is $\Delta(X, Y) = \frac{1}{2}\sum_{v \in \mathcal{X}} |\Pr[X = v] - \Pr[Y = v]|$. Moreover, we say that X and Y are* ε-close *if $\Delta(X, Y) \leq \varepsilon$.*

Next, we define *min-entropy* and *average min-entropy*, the latter introduced by Dodis, Reyzin, and Smith [DRS04]. Average min-entropy captures the remaining unpredictability of X conditioned on the value of Y.

Definition 27. *The* min-entropy *of a random variable X is*

$$H_\infty(X) = -\log(\max_x \Pr[X = x]).$$

The average min-entropy *of random variable X conditioned on random variable Y is*

$$\widetilde{H}_\infty(X|Y) = -\log\left(\mathbb{E}_{y \leftarrow_{\$} Y}\left[2^{-H_\infty(X|Y=y)}\right]\right).$$

Now, we present a bound on average min-entropy that was proved in [DRS04]:

Lemma 2. *If Y has 2^r possible values and Z is any random variable, then $\widetilde{H}_\infty(X|(Y, Z)) \geq \widetilde{H}_\infty(X|Z) - r$.*

In our IKEM construction, we need to derive truly uniform bits from a weakly random source X. In general, this can be done using any *strong randomness extractor* [Sha02]. However, following [PW08], we use pairwise-independent hash functions, which interact particularly well with the notion of average min-entropy. We use the following lemma from [DRS04]:

Lemma 3. *Let X, Y be random variables such that $X \in \{0,1\}^n$ and $\widetilde{H}_\infty(X|Y) \geq k$. Let $\mathcal{H}$ be a family of pairwise-independent hash functions from $\{0,1\}^n$ to $\{0,1\}^\ell$. Then for $h \leftarrow_\$ \mathcal{H}$, we have*

$$\Delta((Y, h, h(X)), (Y, h, U_\ell)) \leq \varepsilon_{\mathcal{H}},$$

as long as $\ell \leq k - 2\log(1/\varepsilon_{\mathcal{H}})$, where U_ℓ is the uniform distribution on $\{0,1\}^\ell$.

B Lower Bound on State Size

In this section, we provide a lower bound showing that after T_1 key regenerations and deletions on inputs $\ell_1, \ldots, \ell_d$ such that $\ell_1 + \cdots + \ell_d = T_0 \leq T_1$, the size of the secret state of any IKEM scheme must be $\Omega((T_1 - T_0) \cdot \lambda)$, where λ is the security parameter. Importantly, this also holds for the (stronger) IKMR notion. Intuitively, the reason for this is that the T_0-th execution of IKM.re-gen must output a fresh, independent public key pk_{T_0} and therefore also add fresh, independent key material to the secret state st_{t_0,T_0}. Moreover, after IKM.re-gen has been executed several more times (and perhaps IKM.del has been executed some number of times), resulting in new state st_{T_0,T_1}, this state must still be able to decapsulate ciphertexts created using pk_{T_0}. Furthermore, again, the key materials added to the state for epochs $T_0 + 1, \ldots, T_1$ must be independent of that which was added in epoch T_0. In combination, the key material for epoch T_0 can not be combined with the key material for epochs $T_0 + 1, \ldots, T_1$; also the key material for epoch $T_0 + 1$ can not be combined with that of $T_0 + 2, \ldots, T_1$; and so on. We now provide the formal theorem and proof.

Theorem 6. *Let* IKM *be any IKEM scheme $(T, \varepsilon^{\mathrm{OW}}_{\mathrm{IKM}})$-secure in the* $\mathrm{OW}_{\mathrm{IKM}}$ *game of Figure 2. For any $\lambda, \alpha, T_1, \ell_1, \ldots, \ell_d \in \mathbb{N}$, $(st_{t_0,t_1}, pk_{t_1}) \leftarrow_\$$* IKM.gen(1^λ) *with $t_0, t_1 \leftarrow 0$ initially, and any sequence of executions of (interspersed in any order) $(st_{t_0,t_1+1}, pk_{t_1+1}) \leftarrow_\$$* IKM.re-gen(st_{t_0,t_1}) *for $t_1 \in [T_1]$ and $st_{t_0+\ell_i,t_1} \leftarrow$* IKM.del$(st_{t_0,t_1}, \ell_i)$ *for $i \in [d]$ (incrementing t_0, t_1 appropriately each time), such that it always holds that $t_0 + \ell_i < t_1$ and $\sum_{i=1}^d \ell_i = T_0$, let st_{T_0,T_1} be the final state. Then $\mathbb{E}[|st_{T_0,T_1}|] \geq (T_1 - T_0) \cdot (\lg \alpha - \alpha \cdot \varepsilon^{\mathrm{OW}}_{\mathrm{IKM}})$.*

Proof. To prove this theorem, we will use st_{T_0,T_1} to encode a random α-ary string $s \in [\alpha]^{T_1 - T_0}$, along with a helper string h (defined below). By Shannon's Source Coding Theorem, if st_{T_0,T_1} and h are uniquely decodable to s, then $\mathbb{E}[|st_{T_0,T_1}|] \geq (T_1 - T_0) \cdot \log \alpha - \mathbb{E}[|h|]$.

Public Randomness. Let $(st_{0,0}, pk_0) \leftarrow_\$$ IKM.gen(1^λ) and let st_{t_0,T_0} be the state after the first T_0 executions of $(st_{t_0,t_1+1}, pk_{t_1+1}) \leftarrow_\$$ IKM.re-gen(st_{t_1}) (with the specified executions of $st_{t_0+\ell_i,t_1} \leftarrow$ IKM.del(st_{t_0,t_1}, ℓ_i) interspersed). Also, let

$$r_{1,0}, r_{1,1}, \ldots, r_{1,\alpha}, r_{2,0}, r_{2,1}, \ldots, r_{2,\alpha}, \ldots, r_{T_1-T_0,0}, r_{T_1-T_0,1}, \ldots, r_{T_1-T_0,\alpha} \leftarrow_\$ \mathcal{R}$$

be re-generation randomness and $\rho \leftarrow \mathcal{R}$ be encapsulation randomness.

Encoding. Alice receives as input random α-ary string $s \leftarrow_\$ [\alpha]^{T_1 - T_0}$. Let $s_t \in [\alpha]$ be the t-th symbol of s for $t \in [T_1 - T_0]$.

1. For $t \in [T_1 - T_0]$:
   (a) For $a \in [\alpha]$: Alice re-generates $(st^a_{t_0,T_0+t}, pk^a_{t_0,T_0+t}) \leftarrow$ IKM.re-gen$(st_{t_0,T_0+t-1}; r_{t,a})$, then runs the specified executions of $(st^a_{t_0+\ell_i,T_0+t}) \leftarrow$ IKM.del$(st^a_{t_0,T_0+t}, \ell_i)$ in the sequence, before the next regeneration.
   (b) Alice sets $st_{t_0,T_0+t} \leftarrow st^{s_t}_{t_0,T_0+t}$.
2. Alice sets $h \leftarrow \epsilon$
3. For $t \in [T_1 - T_0]$:
   (a) For $a \in [\alpha] \setminus \{s_t\}$: Alice encrypts $(k_a, c_a) \leftarrow$ IKM.enc$(pk^a_{t_0,T_0+t}; \rho)$
   (b) If for any $a \in [\alpha] \setminus \{s_t\}$, $k_a =$ IKM.dec(st_{T_0,T_1}, c_a), Alice appends $h \leftarrow (h, s_t)$ (representing s_t using $\log \alpha$ bits).
4. Alice sends to Bob (st_{T_0,T_1}, h).

Decoding. Bob initializes $s' \leftarrow 0^{T_1-T_0}$ and $j \leftarrow 1$. For $t \in [T_1 - T_0]$:

1. For $a \in [\alpha]$:
   (a) Bob re-generates $(st^a_{t_0,T_0+t}, pk^a_{t_0,T_0+t}) \leftarrow$ IKM.re-gen$(st_{t_0,T_0+t-1}; r_{t,a})$ and runs the specified executions of $(st^a_{t_0+\ell_i,T_0+t}) \leftarrow$ IKM.del$(st^a_{t_0,T_0+t}, \ell_i)$ in the sequence, before the next regeneration.
   (b) Next, Bob encapsulates $(k_a, c_a) \leftarrow$ IKM.enc$(pk^a_{t_0,T_0+t}; \rho)$.
   (c) Then, Bob tries to decrypt $k'_a \leftarrow$ IKM.dec(st_{T_0,T_1}, c_a).
2. If $\exists a_1 \neq a_2 \in [\alpha]$ s.t. $k'_{a_1} = k_{a_1}$ and $k'_{a_2} = k_{a_2}$, Bob sets $s'_t \leftarrow h_j$ and increments $j \leftarrow j + 1$; otherwise Bob sets $s'_t \leftarrow a$, for a s.t. $k'_a = k_a$.
3. Finally, Bob sets $st_{t_0,T_0+t} \leftarrow st^{s'_t}_{t_0,T_0+t}$.

Bob then outputs s'.

Analysis. It is clear that Bob will always output the correct string $s' = s$. For $t \in [T_1 - T_0 + 1]$, Alice accounts for the case in which there is some $a \in [\alpha] \setminus \{s_t\}$ such that decapsulated key $k'_a = k_a$, by in Step 3b of the encoding algorithm, appending the correct symbol s_t to h, which is sent to Bob, who will then use it to determine the correct symbol s_t. Otherwise, Bob will correctly set s_t to be the bit a s.t. $k'_a = k_a$, which must exist by correctness since in the encoding algorithm t_1 starts out as $t_1 \geq T_0$ and ends as T_1.

Now, we will analyze how long h will be in expectation, which will tell us, by Shannon's Source Coding Theorem, how long st_{T_0,T_1} must be in expectation. Fix some $t \in [T_1 - T_0]$. We will first bound the probability that for some given $a \in [\alpha] \setminus \{s_t\}$, $k_{at} =$ IKM.dec(st_{T_0,T_1}, c_a) by presenting an attack against $\varepsilon^{\mathrm{OW}}_{\mathrm{IKM}}$-secure scheme IKM by an adversary $\mathcal{A}$ described below:

– After receiving pk_0 from the challenger, $\mathcal{A}$ queries **Re-Gen**() T_0+t-1 times, with the specified executions of **Del**(ℓ_i) interspersed.
– $\mathcal{A}$ then queries **Expose**() to receive st_{t_0,T_0+t-1} and again queries **Re-Gen**() to receive pk_{T_0+t}, followed by the specified executions of **Del**(ℓ_i) in the sequence, before the next regeneration.
– Next, $\mathcal{A}$ queries **Chall**() and receives c^*.

– For $t' \in [T_1 - (T_0 + t - 1)]$, $\mathcal{A}$ samples random re-generation randomness $r_{t'} \leftarrow \mathcal{R}$ and computes $st_{t_0,T_0+t-1+t'} \leftarrow \text{IKM.re-gen}(st_{t_0,t_0+t-1+t'-1}; r_{t'})$, with the specified executions of $\mathbf{Del}(\ell_i)$ interspersed.
– Finally, $\mathcal{A}$ computes $k' \leftarrow \text{IKM.dec}(st_{T_0,T_1}, c^*)$ and forwards k' to its challenger.

Recall it must be that $\varepsilon^{\text{OW}}_{\text{IKM}} \geq \text{Adv}(\mathcal{A}) = \Pr[\mathcal{A} \to k]$. Now, define D_t as the event that given randomly generated pk_{T_0+t}, $(k, c) \leftarrow_\$ \text{IKM.enc}(pk_{T_0+t})$, and st_{T_0,T_1} as above, $k = \text{IKM.dec}(st_{T_0,T_1}, c)$. It is easy to see that $\Pr[\mathcal{A} \to k] = \Pr[D_t]$, and so $\Pr[D_t] \leq \varepsilon^{\text{OW}}_{\text{IKM}}$.

Now note that event D_t also corresponds exactly to the case in which during Alice's encoding algorithm, for some given $a \in [\alpha] \backslash \{s_t\}$, $k_a = \text{IKM.dec}(st_{T_0,T_1}, c_a)$. If we let D^{any}_t be the event in which for *any* $a \in [\alpha] \backslash \{s_t\}$, $k_a = \text{IKM.dec}(st_{T_0,T_1}, c_a)$, then taking the union bound, it can easily be seen that $\Pr[D^{\text{any}}_t] \leq \varepsilon^{\text{OW}}_{\text{IKM}} \cdot \alpha$. So, $\mathbb{E}[|h|] = \sum_{t=1}^{T_1-T_0} \mathbf{1}_{D^{\text{any}}_t} \cdot \Pr[D^{\text{any}}_t] \leq (T_1 - T_0) \cdot \varepsilon^{\text{OW}}_{\text{IKM}} \cdot \alpha$, where $\mathbf{1}_{D^{\text{any}}_t}$ is the indicator random variable taking value 1 when D^{any}_t occurs and 0 otherwise. Therefore, by Shannon's Source Coding Theorem, $\mathbb{E}[|st_{T_0,T_1}|] \geq (T_1 - T_0) \cdot \log \alpha - (T_1 - T_0) \cdot \varepsilon^{\text{OW}}_{\text{IKM}} \cdot \alpha$. $\square$

Corollary 4. *In the statement of Theorem 6, let* $\alpha = 1/\varepsilon^{\text{OW}}_{\text{IKM}}$*. Then* $\mathbb{E}[|st_{T_0,T_1}|] \geq (T_1 - T_0) \cdot (\log(1/\varepsilon^{\text{OW}}_{\text{IKM}}) - 1)$*. In particular, if* $\varepsilon^{\text{OW}}_{\text{IKM}} \leq 2^{-\lambda}$*, then* $\mathbb{E}[|st_{T_0,T_1}|] \geq (T_1 - T_0) \cdot (\lambda - 1)$*.*

C Modified RCCA IKEMR Security

The main problem with defining active security for IKEMR is caused by re-encapsulations: to model active security in the sense of Chosen Ciphertext Attacks (CCA), the security experiment provides a decapsulation oracle that decapsulates adversarial ciphertexts with the victim's secret key. To define security meaningfully and satisfiably, this decapsulation oracle must reject queries for the *challenge ciphertext*—otherwise the challenge is trivially solvable; since the adversary can re-encapsulate challenge ciphertexts with the victim's (newer) public keys, the decapsulation oracle also needs to reject such re-encapsulations of the challenge.

Detecting Trivial Attacks As security definitions for Proxy Re-Encryption (PRE) [BBS98] have to solve the exact same problem, we provide a list of known approaches for detecting and rejecting challenge re-encapsulations in the decapsulation oracle:

1. Only permitting decapsulations of non-challenge ciphertexts that were created by the security experiment itself via a (re-)encapsulation oracle and, thus, directly tracing all relevant re-encapsulations.
Problem: Since, thereby, the adversary can never query the decapsulation oracle on self-created ciphertexts, this approach limits the strength of modeled active attacks significantly.

2. Requiring that constructions offer a predicate for ciphertexts that reveal their re-encapsulation history.
   Problem: Adding such a strong requirement for constructions only to model active attacks limits the applicability of the definition to a reduced selection of constructions.
3. The security experiment detects *deterministic* re-encapsulations with exponential-runtime algorithms [SC09].
   Problem: Security proofs with exponential-runtime reductions are meaningless. This notion also reduces the selection of constructions to those with deterministic re-encapsulation algorithms.
4. Instead of tracing re-encapsulations of the ciphertexts, the security experiment detects and rejects decapsulations of (re-encapsulated) challenges based on their plaintext [CH07, LV08]. This follows the idea of Replayable CCA (RCCA) [CKN03] security.
   Problem: Not only decapsulation queries for re-encapsulations of the challenge ciphertext are rejected, but also other adversarial ciphertext modifications that may not change the underlying plaintext.

Since variant (4) captures the strongest and most natural notion of active attacks for our setting, our security definition is based on the RCCA idea. See also motivation for the RCCA notion in [CKN03], which explains that standard CCA-security is often *too strong* and rules out constructions that intuitively are secure yet cannot achieve CCA-security. Such constructions on the other hand can achieve RCCA-security.

RCCA for KEM-Type Primitives As far as we are aware, RCCA security for KEM-type primitives has not been defined before. When defining RCCA security for a KEM-type primitive, it reasonable to test the validity of this notion by ensuring that it is, indeed, implied by the corresponding CCA security definition and can be used to build IND-RCCA *public key encryption* (PKE) via standard hybrid encryption. For this, we recall that for IND-RCCA security of PKE, the adversary chooses two challenge messages; the decryption oracle then decrypts queried ciphertexts, unless the decrypted message equals either of the two adversarially chosen challenge messages. This condition for rejecting decryption queries is strictly broader than the one for IND-CCA security.

For indistinguishability definitions of KEM-type primitives, the adversary never chooses (or influences) the challenge; instead, the security experiment poses the challenge by either outputting the real encapsulated key or a randomly sampled one. Based on this, it is a priori unclear how an IND-RCCA security definition for a KEM-type primitive would permit and reject decapsulation queries; we see the following options:

1. Reject all decapsulation queries for which the decapsulated key equals the (single) challenge key.
   Problem: This limitation is insufficient because in the *random* world, the adversary can forward the challenge ciphertext to the decapsulation oracle,

which outputs the *real* key. Thus, the adversary can trivially distinguish the real from the random world.

2. Reject all decapsulation queries for which the decapsulated key equals the *real* encapsulated key.
   Problem: It is unclear how to simulate the decapsulation oracle with this condition in a reduction to CCA security, where the reduction has only access to the decapsulation oracle of the CCA security experiment—in particular in the *random* world, since the reduction should not know the *real* key. Thus, it seems as if this variant of RCCA security is not implied by CCA security.
3. Instead of only adapting the decapsulation oracle, we change the form of the posed challenge: as opposed to only either giving the real key k_0 or a random key k_1 as a challenge to the adversary, the adversary receives both keys in two different orders (real world: (k_0, k_1); random world: (k_1, k_0)). The decapsulation oracle is then identical to the PKE setting: all queries are permitted except for those that yield one of the two challenge keys.
   As we will prove next, this notion of IND-RCCA$'$ is implied by IND-CCA security and can also be used to build IND-RCCA-secure PKE via standard hybrid encryption. Furthermore, we show that such a modified challenge format for CCA security, which we call IND-CCA$'$, is equivalent to standard IND-CCA.

Formal Consideration We now give some formal justification for this kind of RCCA for KEM-type primitives; in particular by examining our RCCA IKEMR security notion, which in this section we call IND-RCCA$'_{\mathrm{IKMR}}$. First, for a corresponding notion of CCA security for IKEM, IND-CCA$'_{\mathrm{IKM}}$, we show that IND-CCA$'_{\mathrm{IKM}}$ $\Leftrightarrow$ IND-CCA$_{\mathrm{IKM}}$ and IND-CCA$'_{\mathrm{IKM}}$ $\Rightarrow$ IND-RCCA$'_{\mathrm{IKM}}$. These first two results also show in particular that IND-CCA$'_{\mathrm{KM}}$ $\Leftrightarrow$ IND-CCA$_{\mathrm{KM}}$ and IND-CCA$'_{\mathrm{KM}}$ $\Rightarrow$ IND-RCCA$'_{\mathrm{KM}}$, for basic KEMs. Thus, formally speaking, the modified CCA definition is equivalent to the normal definition for IKEM and basic KEM, and moreover, as is desired, modified RCCA is weaker than modified CCA for the same. Finally, we also show that IND-RCCA$'_{\mathrm{IKMR}}$ + IND-RCCA$_{\mathrm{SK}}$ $\Rightarrow$ IND-RCCA$_{\mathrm{IPKR}}$, where IND-RCCA$_{\mathrm{SK}}$ is RCCA security for symmetric-key encryption and IPKR is an encryption analogue of IKMR. Thus, a primary use case, RCCA-secure interval public key encryption with re-encryptions, can be built from RCCA$'$-secure IKEMR. This also shows that IND-RCCA$'_{\mathrm{KM}}$ + IND-RCCA$_{\mathrm{SK}}$ $\Rightarrow$ IND-RCCA$_{\mathrm{PK}}$.

C.1 CCA$'$ Security is Equivalent to CCA Security for IKEM

We show that IND-CCA$'_{\mathrm{IKM}}$ $\Leftrightarrow$ IND-CCA$_{\mathrm{IKM}}$. The IND-CCA$'$ security game is the same as the IND-CCA game in Figure 2 except that **Chall**() outputs $((k_b, k_{1-b}), c^*)$ instead of just (k_b, c^*).

Theorem 7. *IKEM construction* IKM *is* $(T, O(\varepsilon))$*-secure in game* IND-CCA$'_{\mathrm{IKM}}$ *if and only if it is* $(T, O(\varepsilon))$*-secure in game* IND-CCA$_{\mathrm{IKM}}$.

Proof (sketch). We start by showing the only if direction. A reduction from IND-CCA security to IND-CCA$'$ security, simply forwards all queries from the IND-CCA adversary to the IND-CCA$'$ challenger and forwards the responses back to the adversary exactly the same except for the **Chall**() responses. For **Chall**() responses, the reduction only forwards the first key it receives to the IND-CCA attacker (i.e., k_b). It is clear to see that this simulates exactly the IND-CCA game.

Now we show the if direction. For this we proceed in hybrids. In the first hybrid, the challenge bit of the IND-CCA$'$ game is $b = 0$; thus, the first key output by **Chall**() is the real encapsulated key, and the second key is a random key. We then proceed to the second hybrid in which both keys output by **Chall** are random. A simple reduction to IND-CCA security showing that these hybrids are indistinguishable simply forwards all queries from the adversary to the IND-CCA challenger and forwards the responses back to the adversary exactly the same except for the **Chall**() responses. For **Chall**() responses, the reduction sets k_0 to be the returned key from the IND-CCA challenger, samples random key k_1, and finally forwards (k_0, k_1) to the adversary. If the challenge bit b of the IND-CCA game is $b = 0$ it is clear that this reduction simulates exactly the first hybrid; if $b = 1$ it is clear that this reduction simulates exactly the second hybrid.

In the third hybrid, the first key output by **Chall**() is random, and the second key is the real encapsulated key. A similar reduction to above shows that these hybrids are indistinguishable. Moreover, this third hybrid corresponds exactly to the IND-CCA$'$ game in which the challenge bit is $b = 1$. Thus we are done. □

C.2 CCA$'$ Security Implies RCCA$'$ Security for IKEM

We now show that IND-CCA$'_{\mathrm{IKM}} \Rightarrow$ IND-RCCA$'_{\mathrm{IKM}}$. The IND-RCCA$'_{\mathrm{IKM}}$ security game is the same as the IND-CCA$'_{\mathrm{IKM}}$ security game, except that **Dec**(c) outputs $k \leftarrow \mathrm{IKM.dec}(sk, c)$ unless $k \in \{k_0, k_1\}$.

Theorem 8. *If IKEM construction* IKM *is* (T, ε)*-secure in game* IND-CCA$'_{\mathrm{IKM}}$ *then it is* (T, ε)*-secure in game* IND-RCCA$'_{\mathrm{IKM}}$.

Proof (sketch). A reduction from IND-RCCA$'$ security to IND-CCA$'$ security simply forwards all queries from the IND-RCCA$'$ adversary to the IND-CCA$'$ challenger and forwards the responses back to the adversary exactly the same except for the **Dec**(c) responses. For **Dec**(c) responses, the reduction only forwards the key k it receives to the challenger if $k \notin \{k_0, k_1\}$. It is clear to see that this simulates exactly the IND-RCCA$'$ game. □

C.3 RCCA$'$-secure IKEMR and RCCA-secure SKE implies RCCA-secure IPKER

Finally, we show that IND-RCCA$'_{\mathrm{IKMR}}$ + IND-RCCA$_{\mathrm{SK}} \Rightarrow$ IND-RCCA$_{\mathrm{IPKR}}$ using standard hybrid encryption. IPKR and IND-RCCA$_{\mathrm{IPKR}}$ security are defined in the natural way: (i) There is an encryption algorithm that encrypts a message m instead of encapsulating a key k, a decryption algorithm that decrypts a ciphertext to message m instead of decrypting it to key k, and all other algorithms are unchanged from IKMR; (ii) The $\mathbf{Chall}(pub, m_0, m_1)$ oracle takes in two messages and chooses a random one to encrypt, and only outputs the challenge ciphertext c^* if $pub = 1$ (similarly for **Re-Enc-Chall**), the $\mathbf{Dec}(c)$ oracle decrypts c to m and outputs it only if it is not m_0 or m_1, and all other oracles are unchanged from $\text{IND-RCCA}'_{\text{IKMR}}$.

Hybrid construction We build the hybrid construction for IPKR in the standard way. IPKR.gen and IPKR.re-gen are equivalent to that of IKMR. IPKR.enc(pk, m) computes IKMR.enc$(pk) \to_\$ (k, c_1)$ then uses k to compute SK.enc$(k, m) \to_\$ c_2$ and outputs (c_1, c_2). IPKR.re-enc$(pk, (c_1, c_2))$ simply computes IKMR.re-enc$(pk, c_1) \to_\$ c'_1$ and outputs (c'_1, c_2). Finally, IPKR.dec$(sk, (c_1, c_2))$ simply computes IKMR.dec$(sk, c_1) \to k$ and outputs SK.dec(k, c_2). Correctness of this scheme follows directly from that of IKMR and *SK*. For security, we prove the following theorem.

Theorem 9. *If IKEMR construction* IKMR *is* $(T, \varepsilon_{\text{IKMR}})$*-secure in game* $\text{IND-RCCA}'_{\text{IKMR}}$ *and SKE construction SK is* $(T, \varepsilon_{\text{SK}})$*-secure in game* $\text{IND-RCCA}_{\text{SK}}$*, then the hybrid construction above is* $(T', \varepsilon_{\text{IKMR}} + \varepsilon_{\text{SK}} + (q + 1) \cdot 1/2^\lambda)$*-secure for* $T' \approx T$*, where* q *is the number of queries to the* $\mathbf{Dec}()$ *oracle.*

Proof (sketch). For this proof, we briefly sketch a proof similar to that of [CKN03, Theorem 21]. Given an adversary $\mathcal{A}$ attacking the hybrid scheme, we build a reduction to the IND-RCCA′ security of IKMR. This reduction forwards all queries between the adversary and the IKMR challenger unchanged for all oracles except **Chall**, **Re-Enc-Chall**, **Dec**. For $\mathbf{Chall}(pub, m_0, m_1)$ and $\mathbf{Re\text{-}Enc\text{-}Chall}(pub)$ queries, the reduction queries its own oracles with the same pub; if $pub = 1$, it receives back $((k_b, k_{1-b}), c_1^*)$. Then, it flips its own coin $\delta \leftarrow_\$ \{0, 1\}$ and uses always k_b to encrypt SK.enc$(k_b, m_\delta) \to_\$ c_2^*$ and sends (c_1^*, c_2^*) to $\mathcal{A}$. Finally, for $\mathbf{Dec}((c_1, c_2))$ queries, the reduction queries its own oracle on c_1 to receive back k; if $k = \perp$, the reduction returns SK.dec$(k_b, c_2) \to m$ to $\mathcal{A}$, otherwise, it returns SK.dec$(k, c_2) \to m$ (in both cases, if m is not m_1 or m_2). When the adversary returns bit δ', the reduction outputs 0 if $\delta' = \delta$ and 1 otherwise.

The proof concludes with two claims. In the first claim, we argue that if the IKMR challenge ciphertext c_1^* encapsulates k_b, then the reduction simulates a statistically-close world to the real IPKR game for $\mathcal{A}$. Indeed, the only way the reduction differs from the real game is if $\mathcal{A}$ queries $\mathbf{Dec}((c_1, c_2))$, where c_1 encapsulates k_{1-b}. In this case, the reduction responds with SK.dec(k_b, c_2), but the real game responds with SK.dec(k_{1-b}, c_2). However, this event only happens with probability $1/2^\lambda$ for each **Dec** query, since k_{1-b} is random and independent of the view of $\mathcal{A}$.

In the second claim, we argue that if the IKMR challenge ciphertext c_1^* encapsulates k_{1-b}, then the advantage of $\mathcal{A}$ is bounded by ε_{SK}. Intuitively, this is because the key used to encrypt c_2^*, k_b, is independent of everything else, as long as $k_b \neq k_{1-b}$, which only happens with probability $1/2^\lambda$. Thus, we can directly reduce to the security of SK.

Finally, the theorem statement follows from these two claims by observing that the advantage of $\mathcal{A}$ in these two worlds must be negligibly-close. Indeed, since the advantage in the first world is negligibly-close to the advantage of $\mathcal{A}$ against the IND-RCCA$_{\text{IPKR}}$ security of the hybrid scheme, and the advantage in the second world is negligibly small, it must be that the advantage of $\mathcal{A}$ against the IND-RCCA$_{\text{IPKR}}$ security of the hybrid scheme is negligibly small. See [CKN03, Theorem 21] for more details. □

D Missing Proofs

Theorem 10 (Theorem 1, restated). *Let* $X \in \{\text{CPA}, \text{CCA}\}$ *and* KM *be a* $(T, \varepsilon^{\text{ind-x}}_{\text{KM}})$*-secure KEM scheme in the* IND-X *KEM security game. Then the* IKM *construction of Figure 3 is correct and* $(T', T \cdot \varepsilon^{\text{ind-x}}_{\text{KM}})$*-secure, for* $T' \approx T$*, in game* IND-X$_{\text{IKM}}$ *of Figure 2.*

Proof. Correctness is immediate since the receiver will store the KEM secret keys for all of the epochs between t_0 and t_1 in ST. For security, we reduce directly to that of the KM scheme. For the epoch in which $\mathcal{A}$ queries **Chall**(), it is clear that the corresponding KEM secret key sk will be deleted from ST before the next (successful) **Expose**() query, since **Expose** is only permitted if $t^* \geq t_0$. So, $\mathcal{A}$ learns nothing about sk and thus the challenge ciphertext must be secure.

More formally, we define our reduction algorithm $\mathcal{R}$ attacking scheme KM as follows. We prove more challenging CCA security first. First, $\mathcal{R}$ receives from its challenger (pk^*, k^*, c^*). Then, $\mathcal{R}$ guesses the challenge epoch $t^* \leftarrow_\$ [0, T]$ in which $\mathcal{A}$ will query **Chall**(). For (possibly) initialization and every **Re-Gen**() query for epoch $t_1 \neq t^*$, $\mathcal{R}$ samples its own KM key pair $(sk, pk) \leftarrow_\$ \text{KM.gen}(1^\lambda)$ and follows the steps of IKM.gen() or IKM.re-gen() as specified, then outputs (pk, t_1). For initialization if $t^* = 0$ or the **Re-Gen**() query for epoch t^*, $\mathcal{R}$ uses the public key pk^* received from its challenger, sets $sk \leftarrow \perp$ and otherwise follows IKM.gen() or IKM.re-gen() as specified, then outputs (pk^*, t^*). For every **Del**(ℓ) query, $\mathcal{R}$ follows the steps of IKM.del(ℓ) as specified.

For the **Chall**() query, if it comes during an epoch $t \neq t^*$, $\mathcal{R}$ simply outputs random bit b' to its challenger. Otherwise, (if it is a valid challenge) $\mathcal{R}$ simply returns the challenge key k^* and ciphertext c^* (from its challenger), appended with t^* to $\mathcal{A}$. For **Expose**() queries, if **Chall** has not been queried yet or $t_0 > t^*$, $\mathcal{R}$ directly outputs (ST, t_0, t_1); otherwise outputs $\perp$.

Finally, $\mathcal{R}$ answers **Dec**$((c, t))$ queries from $\mathcal{A}$ as follows. If $t = t^*$ then $\mathcal{R}$ forwards c to its challenger and forwards the response back to $\mathcal{A}$. Otherwise, if $t \in [t_0, t_1]$, $\mathcal{R}$ returns IKM.dec$(ST[t], c)$ to $\mathcal{A}$; else, it returns $\perp$

Ultimately (if $\mathcal{R}$ has not already output its own random guess bit to its challenger), $\mathcal{R}$ forwards the bit b' received from $\mathcal{A}$ to its challenger.

If $\mathcal{R}$ guesses t^* correctly, it is clear that $\mathcal{R}$ simulates the real IKM security game exactly, where the challenge bit b of the KM security game decides the challenge bit of the IKM security game. Indeed, all public keys and the challenge ciphertext are generated identically and for **Expose**(), since if $t^* \geq t_0$ the state is never output, the dictionary entry with $sk = \bot$ will never be returned to $\mathcal{A}$. Furthermore, decryptions for all ciphertexts $(c, t), t \neq t^*$, are directly handled by $\mathcal{R}$. For those in which $t = t^*$, if $c = c^*$, the challenger, and thus $\mathcal{R}$, will output $\bot$ as in the IKM game; otherwise, decryption will will be simulated perfectly.

Thus, IKM from Figure 3 is correct and $(T', T \cdot \varepsilon_{\text{KM}}^{\text{ind-cca}})$-secure in game IND-CCA$_{\text{IKM}}$ of Figure 2. IND-CPA security from $(T, \varepsilon_{\text{KM}}^{\text{ind-cpa}})$-secure KM follows directly from reduction $\mathcal{R}$ above, by simply removing the handling of **Dec** queries. □

Theorem 11 (Theorem 2 re-stated). *If* P *is correct, L-lossy, and $(T, \varepsilon_{\text{P}})$-secure, and* G *is $(T, \varepsilon_{\text{G}})$-secure then the above* FSP *construction is correct, L-lossy, and $(T', \varepsilon_{\text{P}} + T \cdot \varepsilon_{\text{G}})$-secure, for $T' \approx T$.*

Proof. Correctness is clear since for any $t_0 \leq t_1$, SK_{t_0} will still store s_{t_0} which can be exapnded to sk_{t_1}, and the P for each epoch is in injective mode $(b = 1)$, so the computation of P.inv(sk_{t_1}, y) will be successful. FSP is L-lossy since when $b = 0$, the P for epoch t^* is in lossy mode, and is thus L-lossy.

For security, we first reduce to the security of G. Indeed, consider hybrids H$_i$ for $i \in [0, T]$ in which when computing the seed s_{t^*+1} from which the seed $s_{t'}$ that is given to the adversary for $SK_{t'}$ is computed, we sample the j-th seed for $j = \min\{i, t^* + 1\}$, $s_j \leftarrow_\$ \mathcal{S}$ randomly (and continue computing the future seeds as normal, using G). Thus, H$_0$ is the real security game. It is clear that for any adversary $\mathcal{A}$, the differences of advantage of $\mathcal{A}$ between hybrids H$_{i-1}$ and H$_i$ for $i \in [T]$ is at most ε_{G}; namely $T \cdot \varepsilon_{\text{G}}$ between H$_0$ and H$_T$.

Next, we reduce directly to the security of P to show that the adversary's advantage in hybrid H$_T$ is at most ε_{P}. If $\mathcal{A}$ is given $SK_{t'}$ and $t' > t^*$, then $SK_{t'}$ only contains $s_{t'}$ that is computed from random seed s_{t^*+1}, which is independent from sk_{t^*} and so we get security directly from that of the t^*-th trapdoor permutation. More formally, we define reduction algorithm $\mathcal{R}$ as follows. $\mathcal{R}$ first samples $(sk_t, pk_t) \leftarrow_\$ \text{P.gen}(1^\lambda)$ for all $t \in [t' - 1] \setminus \{t^*\}$ and random $s_{t^*+1} \leftarrow_\$ \mathcal{S}$, then computes (sk_t, pk_t) for $t \in [t^* + 1, f - 1]$ as normal in the FSP construction, using G iteratively on s_{t^*+1}. Then, it receives pk from its challenger, sets $pk_{t^*} \leftarrow pk$ and sends to $\mathcal{A}$: $(\{pk_0, \ldots, pk_{f-1}\})$. Upon reception of $t^* < t' \leq f$ from $\mathcal{A}$, $\mathcal{R}$ sends to $\mathcal{A}$: $(t', s_{t'})$. $\mathcal{R}$ then forwards the bit b' received from $\mathcal{A}$ to its challenger. It is clear that $\mathcal{R}$ simulates H$_T$ perfectly for $\mathcal{A}$ and so $(T', \varepsilon_{\text{P}})$-security of H$_T$ follows from that of P.

Therefore, $(T', \varepsilon_{\text{P}} + T \cdot \varepsilon_{\text{G}})$-security of FSP follows from that of P and G. □

Theorem 12 (Theorem 3 re-stated). *Let* FSP *be a family of correct, L-lossy, and $(T, \varepsilon_{\text{FSP}})$-secure trapdoor permutations on common domain $\mathcal{X} = \{0,1\}^n$; let $\mathcal{H}_2 : \{0,1\}^n \to \{0,1\}^\ell$ be a family of pairwise independent hash functions where $\ell \leq k - 2\log(1/\varepsilon_{\mathcal{H}})$, for some $k = \omega(\log n)$ and some negligible $\varepsilon_{\mathcal{H}_2} = \mathsf{negl}(\lambda)$, where $\log(L) + \log(L') \geq n + k$;* OTS *be a strongly unforgeable one-time signature scheme where the verification keys are in $\{0,1\}^v$; and* ABO *be a family of correct, L'-lossy, and $(T, \varepsilon_{\text{ABO}})$-secure all-but-one trapdoor functions* *on domain* $\{0,1\}^n$. *Then, for* $T' \approx T$, *the* IKMR *construction of Figure 5 is correct and* $((T', T^2 \cdot (\varepsilon_{\text{OTS}} + 2 \cdot (O(1/2^\lambda) + \varepsilon_{\text{ABO}} + \varepsilon_{\text{FSP}}) + \varepsilon_{\mathcal{H}_2}))$*-secure) in game* IND-RCCA$_{\text{IKMR}}$ *of Figure 4.*

Proof. Correctness is immediate from that of the FSP family and the OTS scheme, and since ABO is deterministic, as well as the facts that (i) the receiver stores the secret keys of epochs $t' \in [t_0, t_1]$ in SK with remaining FSP epochs corresponding to $t_0, t_0+1, \ldots, t'$, respectively, and also (ii) that ciphertexts label the intervals of epochs whose public keys were used to permute c. Indeed, these secret keys and intervals can then be used by the receiver to correspondingly invert c and hash the last inverted element x to get the key $k = h(x)$.

For security, we will proceed by a sequence of hybrids. Let H_0^A be the real (adaptive) game of Figure 4. H_0^S is the *selective* version of the same. That is, at the beginning of the game, the adversary $\mathcal{A}$ specifies the epoch t^* in which it will query **Chall**() and the epoch t_{pub} in which it queries either **Chall**(pub) or **Re-Enc-Chall**(pub) with $pub = 1$ (all other queries can be made adaptively). We show first that for any adversary $\mathcal{A}$ against game H_0^A, there exists reduction $\mathcal{R}$ against game H_0^S such that $\Pr[b \leftarrow_\$ \text{H}_0^S(\mathcal{R})] \geq \frac{1}{T^2} \cdot \Pr[b \leftarrow_\$ \text{H}_0^A(\mathcal{A})] + \frac{T^2-1}{2T^2}$. $\mathcal{R}$ proceeds simply as follows, emulating H_0^A for $\mathcal{A}$. At the beginning of the game, $\mathcal{R}$ guesses $t^* \leq t_{pub} \leftarrow_\$ [T]^2$, and sends them to H_0^S. Then, for every query sent from $\mathcal{A}$, $\mathcal{R}$ forwards it to H_0^S. However, if $\mathcal{A}$ does not query **Chall**() in epoch t^*, or the epoch in which $\mathcal{A}$ queries **Chall**(pub) or **Re-Enc-Chall**(pub) with $pub = 1$ is not t_{pub}, $\mathcal{R}$ outputs random bit b' to the challenger. At the end of the game, $\mathcal{R}$ forwards the guess bit b' from $\mathcal{A}$ to H_0^S. It can easily be seen that the probability that $\mathcal{R}$ guesses t^*, t_{pub} correctly is at least $1/T^2$ and when it does, it has the same probability of guessing the challenge bit b as $\mathcal{A}$; otherwise it guesses b with probability $1/2$.

Now, hybrid $\text{H}_{0,1}^S$ is the same as H_0^S, except at the beginning of initialization, the hybrid samples $(sk^*, vk^*) \leftarrow \text{OTS.gen}(1^\lambda)$ and during the **Chall**() query, the hybrid changes IKMR.enc() by using (sk^*, vk^*) instead of sampling them on its own. It is clear that the view of the adversary in these two hybrids is the same, since **Chall** is only queried once, and thus the probability that the adversary outputs the challenge bit is unchanged.

Hybrid $\text{H}_{0,2}^S$ is the same as $\text{H}_{0,1}^S$, except that if **Dec**(c) is queried with $c = (c_1, vk, c_2, \sigma, ((t_{0,0}, t_{0,1}), \ldots, (t_{l,0}, t_{l,1})))$, where $vk = vk^*$, then $\perp$ is output. Otherwise **Dec**(c) queries are processed as usual. First, observe that $\text{H}_{0,1}^S$ and $\text{H}_{0,2}^S$ behave the same unless event E occurs, where $\mathcal{A}$ makes a **Dec**(c) query in which the included verification key $vk = vk^*$ and the included ABO output $c_2 \neq c_2^*$, and $\text{OTS.ver}(vk^*, c_2, \sigma) = 1$, where σ is the included signature. Indeed, if $vk = vk^*$ and $c_2 = c_2^*$, then $\text{H}_{0,1}^S$ will output $\perp$ since the decapsulated key will be the real challenge key, or c is malformed. This is because ABO is initialized with lossy branch $0^v \neq vk^*$ except with probability $O(1/2^\lambda)$, thus there is a unique x such that $\text{ABO.eval}(pk', vk^*, x) = c_2^*$, and since the IKMR.dec algorithm checks that the found value x' satisfies $\text{ABO.eval}(pk', vk^*, x') = c_2^*$, it must be that $x' = x$ and $h(x') = h(x)$. We show that the above event, E, happens only with probability ε_{OTS}, and thus for any adversary $\mathcal{A}$, $\Pr[b \leftarrow_{\$} \mathrm{H}^S_{0,1}(\mathcal{A})] \leq O(1/2^\lambda) + \varepsilon_{\mathrm{OTS}} + \Pr[b \leftarrow_{\$} \mathrm{H}^S_{0,2}(\mathcal{A})]$, by constructing reduction $\mathcal{R}$ against the security game of OTS. In particular, $\mathcal{R}$ first receives from $\mathcal{A}$: t^*, t_{pub}; and samples the challenge bit for the IKMR game $b \leftarrow_{\$} \{0,1\}$. $\mathcal{R}$ also receives from the OTS challenger verification key vk^*, instead of sampling $(sk^*, vk^*) \leftarrow \mathrm{OTS.gen}(1^\lambda)$ itself. $\mathcal{R}$ emulates $\mathrm{H}^S_{0,1}$ exactly, until the **Chall** query. For this query, $\mathcal{R}$ proceeds as in IKMR.enc(), except queries the OTS challenger on message c_2^* (the output of ABO) and receives back the signature σ^* that it includes in the challenge ciphertext. Then for any subsequent $\mathbf{Dec}(c)$ query in which vk^* is included as the verification key, $c_2 \neq c_2^*$ is included as the output of ABO, and σ is included as the signature on c_2, if $\mathrm{OTS.ver}(vk^*, c_2, \sigma) = 1$, then $\mathcal{R}$ sends (c_2, σ) to its challenger; otherwise, it outputs $\perp$. It is clear that $\mathcal{R}$ simulates $\mathrm{H}^S_{0,1}$ and $\mathrm{H}^S_{0,2}$ perfectly, up until the point in which it receives a $\mathbf{Dec}(c)$ query in which vk^* is included as the verification key, $c_2 \neq c_2^*$ is included as the ABO output, and the included signature σ successfully verifies. At this point, $\mathcal{R}$ wins the game. Therefore, event E only happens with probability $\varepsilon_{\mathrm{OTS}}$.

Hybrid $\mathrm{H}^S_{0,3}$ is the same as $\mathrm{H}^S_{0,2}$, except that during computation of IKMR.gen in initialization, the ABO function is chosen to have a lossy branch $b^* = vk^*$ rather than $b^* = 0^v$; i.e., $(\cdot, pk') \leftarrow_{\$} \mathrm{ABO.gen}(1^\lambda, 0^v)$ is replaced with $(\cdot, pk') \leftarrow_{\$} \mathrm{ABO.gen}(1^\lambda, vk^*)$. We show that for any adversary $\mathcal{A}$, $\Pr[b \leftarrow_{\$} \mathrm{H}^S_{0,2}(\mathcal{A})] \leq 2 \cdot \varepsilon_{\mathrm{ABO}} + \Pr[b \leftarrow_{\$} \mathrm{H}^S_{0,3}(\mathcal{A})]$, by constructing reduction $\mathcal{R}$ against the security game of ABO. In particular, during initialization, $\mathcal{R}$ sends 0^v and vk^* to its challenger, which returns pk', and $\mathcal{R}$ uses this as the ABO public key thereafter. Otherwise, $\mathcal{R}$ simulates exactly as $\mathrm{H}_{0,2}$ sampling challenge bit for the IKMR game $b \leftarrow_{\$} \{0,1\}$ during the **Chall** query. When $\mathcal{R}$ receives bit b' from $\mathcal{A}$, if $b' = b$, $\mathcal{R}$ sends to its challenger 0; otherwise $\mathcal{R}$ sends 1. Let δ be the challenge bit of the ABO security game. It is clear that if $\delta = 0$, then $\mathcal{R}$ perfectly simulates $\mathrm{H}^S_{0,2}$; otherwise, $\mathcal{R}$ perfectly simulates $\mathrm{H}^S_{0,3}$. Therefore, we have that $\Pr[b \leftarrow_{\$} \mathrm{H}^S_{0,2}(\mathcal{A})] = \Pr_b[\mathcal{A} \to b | \delta = 0]$ and $\Pr[b \leftarrow_{\$} \mathrm{H}^S_{0,3}(\mathcal{A})] = \Pr_b[\mathcal{A} \to b | \delta = 1]$. Thus, $1/2 \cdot (\Pr[b \leftarrow_{\$} \mathrm{H}^S_{0,2}(\mathcal{A})] - \Pr[b \leftarrow_{\$} \mathrm{H}^S_{0,3}(\mathcal{A})]) = 1/2 \cdot (\Pr_b[\mathcal{R} \to 0 | \delta = 0] - \Pr_b[\mathcal{R} \to 0] | \delta = 1]) = \Pr[\mathcal{R} \to \delta : \delta \leftarrow_{\$} \{0,1\}] - 1/2 \leq \varepsilon_{\mathrm{ABO}}$, by the security of ABO.

Hybrid $\mathrm{H}^S_{0,4}$ is the same as $\mathrm{H}^S_{0,3}$, except that the hybrid keeps the ABO secret key when sampling $(sk', pk') \leftarrow_{\$} \mathrm{ABO.gen}(1^\lambda, vk^*)$ during initialization, and for $\mathbf{Dec}(c)$ queries with $c = (c_1, vk, c_2, \sigma, ((t_{0,0}, t_{0,1}), \ldots, (t_{l,0}, t_{l,1}))$, we compute $x \leftarrow \mathrm{ABO.inv}(sk', vk, c_2)$ instead of using FSP to compute x (the signature verification and well-formedness checks for both the ABO and FSP components are kept). Note that $\mathbf{Dec}(c)$ queries in which $vk = vk^*$ are still rejected. We now show that the two hybrids are perfectly equivalent. We can first assume $vk \neq vk^*$ by definition of the hybrids. Now, both hybrids check that $c_2 = \mathrm{ABO.eval}(pk', vk, x)$ and that c_1 is equal to the output of the FSP evaluations specified by $(t_{0,0}, t_{0,1}), \ldots, (t_{l,0}, t_{l,1})$, on input x, for some x that they compute (in different ways), and output $\perp$ if not. Thus, some x exists, and it suffices to show that this x is unique, and both hybrids find it. In both hybrids, each FSP is initialized in injective mode and ABO is initialized with lossy branch vk^*. Therefore, there is a unique x such that $c_2 = \mathrm{ABO.eval}(pk', vk, x)$. Additionally, for every i from l to 0 and every t' from $t_{i,1}$ to $t_{i,0}$, there is a unique $c_1^{t'-1}$ such that $c_1^{t'} = \text{FSP.eval}(PK[t'], t_{0,0} - t'_0, c_1^{t'-1})$, where $c_1^{t_{l,1}} = c_1$. Thus, there is a unique x that results in c_1 when the FSP evaluations specified by $(t_{i,0}, t_{i,1})$ for $i \in [0, l]$ are applied to it. In hybrid $\text{H}^S_{0,3}$, x is found by computing the FSP inversions specified by the $(t_{i,0}, t_{i,1})$; in hybrid $\text{H}^s_{0,4}$, x is found by computing $x \leftarrow \text{ABO.inv}(sk', vk, c_2)$.

Now, hybrid H^S_1 is the same as $\text{H}^S_{0,4}$, except when **Re-Gen**() is queried for the t_{pub}-th time, the execution of FSP.gen is replaced with $\text{FSP.gen}(1^\lambda, t_{pub} - t_0 + 1, 0, t^* - t_0)$ (i.e., the lossy mode for epoch $t^* - t_0$). We show that for any adversary $\mathcal{A}$, $\Pr[b \leftarrow_\$ \text{H}^S_{0,4}(\mathcal{A})] \leq 2 \cdot \varepsilon_{\text{FSP}} + \Pr[b \leftarrow_\$ \text{H}^S_1(\mathcal{A})])$, by constructing reduction $\mathcal{R}$ against the security game of FSP. In particular, $\mathcal{R}$ first receives from $\mathcal{A}$: t^*, t_{pub}; and samples the challenge bit for the IKMR game $b \leftarrow_\$ \{0, 1\}$. $\mathcal{R}$ emulates H^S_0 exactly, until the t_{pub}-th query of **Re-Gen**(). For this query, let t^*_0 be the given the value of t_0 at the time. $\mathcal{R}$ then forwards to its challenger lossy epoch $t^*_{\text{FSP}} = t^* - t_0$ and $f = t_{pub} - t_0 + 1$, and for this t_{pub}-th query of **Re-Gen**(), $\mathcal{R}$ forwards the PK output by the FSP security game to $\mathcal{A}$. Then, $\mathcal{R}$ continues emulating H^S_0 exactly, until the next **Expose**() query, at which point (if it is valid) for t_0 corresponding to current active interval $[t_0, t_1]$, $\mathcal{R}$ sends to the FSP security game exposure epoch $t' \leftarrow t_0 - t^*_0 + 1$. $\mathcal{R}$ receives back $SK_{t'}$, and sets $ST[t_{pub}] \leftarrow SK_{t'}$, before returning ST. $\mathcal{R}$ then emulates H^S_0 for the rest of the game, until $\mathcal{A}$ sends it a guess bit b'. If $b' = b$, $\mathcal{R}$ sends to its challenger 1; otherwise $\mathcal{R}$ sends 0. (Note: all **Dec**(c) queries are answered without using FSP.inv.)

Let δ be the challenge bit of the FSP security game. Then $\Pr[b \leftarrow_\$ \text{H}^S_{0,4}(\mathcal{A})] = \Pr_b[\mathcal{A} \to b | \delta = 1]$ and $\Pr[b \leftarrow_\$ \text{H}^S_1(\mathcal{A})] = \Pr_b[\mathcal{A} \to b | \delta = 0]$. Therefore, $1/2 \cdot (\Pr[b \leftarrow_\$ \text{H}^S_{0,4}(\mathcal{A})] - \Pr[b \leftarrow_\$ \text{H}^S_1(\mathcal{A})]) = 1/2 \cdot (\Pr[\mathcal{R} \to 1 | \delta = 1] - \Pr[\mathcal{R} \to 1 | \delta = 0]) = \Pr[\mathcal{R} \to \delta : \delta \leftarrow_\$ \{0, 1\}] - 1/2 \leq \varepsilon_{\text{FSP}}$, by the security of FSP.

Finally, we show that in H^S_1, any $\mathcal{A}$ (even unbounded) can guess challenge bit b with probability at most $1/2 + \varepsilon_{\mathcal{H}_2}$ advantage, unconditionally. Consider the random variable x sampled for the challenge encapsulation, which is chosen independently of the view $\mathcal{V}$ of $\mathcal{A}$. Since $pk_{t_{pub}}$ is sampled in lossy mode for the challenge epoch in H^S_1, $|\text{FSP.eval}(pk, t^* - t_0, \cdot)| \leq 2^n/L$. Additionally, pk' is sampled with lossy branch vk^*, which means that $|\text{ABO.eval}(pk', vk^*, \cdot)| \leq 2^n/L'$. Therefore, the random variable $(c^*_1, c^*_2) = (\text{FSP.eval}(pk, t^* - t_0, x), \text{ABO.eval}(pk', vk^*, x))$ can take on at most $2^{2n}/(L \cdot L') \leq 2^{n-k}$ values by assumption that $\log(L) + \log(L') \geq n + k$. Thus, by Lemma 2, given the output $c^*_1 \leftarrow \text{FSP.eval}(pk, t^* - t_0, x)$ and $c^*_2 \leftarrow \text{ABO.eval}(pk', vk^*, x)$ of the public challenge ciphertext, $\widetilde{H}_\infty(x | c^*_1, c^*_2, \mathcal{V}) \geq \widetilde{H}_\infty(x | \mathcal{V}) - n + k = k$. Therefore, by Lemma 3 and the hypothesis that $\ell \leq k - 2\log(1/\varepsilon_{\mathcal{H}_2})$, we have that $h(x)$ is $\varepsilon_{\mathcal{H}_2}$-close to uniform (conditioned on the rest of $\mathcal{A}$'s view). This means that the output of **Chall** in this hybrid is $\varepsilon_{\mathcal{H}_2}$-close to (k_b, k_{1-b}) for both k_b, k_{1-b} random, and thus $\mathcal{A}$ can only guess b with advantage $\varepsilon_{\mathcal{H}_2}$.

Thus to conclude, we have shown that for any $\mathcal{A}$, $\Pr[b \leftarrow_\$ \text{H}^A_0(\mathcal{A})] \leq 1/2 + T^2 \cdot (O(1/2^\lambda) + \varepsilon_{\text{OTS}} + 2 \cdot (\varepsilon_{\text{ABO}} + \varepsilon_{\text{FSP}}) + \varepsilon_{\mathcal{H}})$. □

We also show a Theorem for CPA security.

Theorem 13. *Let* FSP *be a family of correct, L-lossy, and $(T, \varepsilon_{\mathrm{FSP}})$-secure trapdoor permutations on common domain $\mathcal{X} = \{0,1\}^n$; and $\mathcal{H}_1 : \{0,1\}^n \to \{0,1\}^\ell$ be a family of pairwise independent hash functions where $\ell \leq \log(L) - 2\log(1/\varepsilon_{\mathcal{H}})$, for some negligible $\varepsilon_{\mathcal{H}_1} = \mathsf{negl}(\lambda)$. Then, for $T' \approx T$, the* IKMR *construction of Figure 5 is correct and $(T', T^2 \cdot (2\varepsilon_{\mathrm{FSP}} + \varepsilon_{\mathcal{H}_1}))$-secure in game* IND-CPA$_{\mathrm{IKMR}}$ *of Figure 4.*

Proof. This follows from the proof of Theorem 3, by simply skipping directly from hybrid H_0^S to hybrid H_1^S via a similar reduction to the security of FSP, and accounting for the different output length of the hash function family. □

Theorem 14 (Theorem 4 re-stated). *Let* H *be a correct and $(T_{\mathrm{H}}, \varepsilon_{\mathrm{H}})$-collision-resistant hash function and* FSE *be a correct and $(T_{\mathrm{FSE}}, \varepsilon_{\mathrm{FSE}})$-secure FS-PKE. Then the* IKMR *construction of Figure 6 is correct and $(T', \varepsilon_{\mathrm{H}} + T^2 \cdot \varepsilon_{\mathrm{FSE}})$-secure in game* IND-RCCA$_{\mathrm{IKMR}}$ *of Figure 4, for $T' \approx T_{\mathrm{H}} + T_{\mathrm{FSE}}$.*

Proof. Correctness is immediate from that of the FSE as well as the facts that (i) the receiver stores the secret keys of epochs $t' \in [t_0, t_1]$ in SK with remaining FSE epochs corresponding to $t_0, t_0 + 1, \ldots, t'$, respectively, and also (ii) that ciphertexts label the epochs for each public key that was used to re-encrypt c. Indeed, these secret keys can then be used by the receiver to correspondingly decrypt c to get the key k.

For security, we will first use the hash function's collision resistance and then proceed by a sequence of hybrids.

In our first game hop we abort if oracles **Chall**, **Re-Enc-Chall**, or **Dec** ever take as input or produce as output ciphertexts that, at any re-encryption level, contain the same hash value for a different list of re-encapsulation epochs. Detecting this modification implies producing a hash collision, which, via a direct reduction, breaks the collision-resistance of the hash function.

Let H_0^A be the real (adaptive) game of Figure 4. H_0^S is the *selective* version of the same. That is, at the beginning of the game, the adversary $\mathcal{A}$ specifies the epoch t^* in which it will query **Chall**() and the epoch t_{pub} in which it queries either **Chall**(pub) or **Re-Enc-Chall**(pub) with $pub = 1$. Using the same steps as in the proof of Theorem 3, we can show that for any adversary $\mathcal{A}$ against game H_0^A, there exists reduction $\mathcal{R}$ against game H_0^S such that $\Pr[b \leftarrow_\$ \mathrm{H}_0^S(\mathcal{R})] \geq \frac{1}{T^2} \cdot \Pr[b \leftarrow_\$ \mathrm{H}_0^A(\mathcal{A})] + \frac{T^2-1}{2T^2}$.

Now, hybrid H_1^S is the same as H_0^S, except in epoch t_{pub} when **Chall**(1) or **Re-Enc-Chall**(1), the execution of FSE.enc replaces input k, resp. c, with a random bit string of the same length. We show that for any adversary $\mathcal{A}$, $\Pr[b \leftarrow_\$ \mathrm{H}_0^S(\mathcal{A})] \leq \varepsilon_{\mathrm{FSE}} + \Pr[b \leftarrow_\$ \mathrm{H}_1^S(\mathcal{A})]$, by constructing reduction $\mathcal{R}$ against the security game of FSE. In particular, $\mathcal{R}$ first receives from $\mathcal{A}$: t^*, t_{pub}. $\mathcal{R}$ emulates H_0^S exactly, until **Chall**(1) or **Re-Enc-Chall**(1) are queried at time t_{pub}. For this query, let t_0^* be the given the value of t_0 at the time. $\mathcal{R}$ then forwards to its challenger epoch $t^*_{\mathrm{FSE}} = t^* - t_0$ as well as the real input k, resp. c, and a randomly sampled string of the same length, and for this query, $\mathcal{R}$ forwards the resulting c output by the FSE security game to $\mathcal{A}$ (after adding the list of encryption layers $(t_0^\circ, \ldots, t_l^\circ)$).

Once $\mathcal{A}$ queries oracle $\mathbf{Dec}(c')$, reduction $\mathcal{R}$ honestly decrypts all layers of FSE encryptions until it reaches a layer $c'_{t_{pub}}$ that is encrypted to epoch t_{pub}; if ciphertext c' contains no layer encrypted to epoch t_{pub}, the entire query to oracle **Dec** is simulated honestly by reduction $\mathcal{R}$. For ciphertext $c'_{t_{pub}}$, reduction $\mathcal{R}$ behaves as follows: (1) if $c \neq c'_{t_{pub}}$ (i.e., the published challenge ciphertext c differs from the layer queried for decryption $c'_{t_{pub}}$), FSE security game's oracle **Dec** is queried for $c'_{t_{pub}}$ to obtain the encrypted payload $m'_{t_{pub}}$; the decryption of $m'_{t_{pub}}$ for the remaining layers is again simulated by $\mathcal{R}$. (2) if $c = c'_{t_{pub}}$ but the encryption layers $(t_0^\circ, \ldots, t_l^\circ)$ of the two ciphertexts differ (i.e., the inner encryption layer structure of the published challenge ciphertext c differs from the one that is queried for decryption of $c'_{t_{pub}}$, but the actual ciphertexts are equal), $\mathcal{R}$ rejects this query based on the first game hop: as the two ciphertexts are the same but the list of encryption layers differ, either the encrypted hash value is a collision (upon which the first game hops rejects already), or the encrypted hash value is not the same as the hash of the actual list of encryption layers. (3) if $c = c'_{t_{pub}}$ and the encryption layers $(t_0^\circ, \ldots, t_l^\circ)$ of the two ciphertexts are the same, $\mathcal{R}$ rejects this query as it will leak the challenge key, which is forbidden by the RCCA game.

$\mathcal{R}$ continues emulating H_0^S exactly, until the next $\mathbf{Expose}()$ query, at which point (if it is valid) for t_0 corresponding to current active interval $[t_0, t_1]$, $\mathcal{R}$ sends to the FSE security game exposure epoch $t' \leftarrow t_0 - t_0^* + 1$. $\mathcal{R}$ receives back $SK_{t'}$, and sets $ST[t_{pub}] \leftarrow SK_{t'}$, before returning ST. $\mathcal{R}$ then emulates H_0^S for the rest of the game, until $\mathcal{A}$ sends it a guess bit b', which $\mathcal{R}$ forwards to its challenger 1.

Finally, we show that in H_1^S, $\mathcal{A}$ can guess challenge bit b with probability at most $1/2$. This is true because the random key k sampled for the challenge encapsulation, is now independent of the published ciphertext seen by $\mathcal{A}$.

Thus to conclude, we have shown that for $\mathcal{A}$, $\Pr[b \leftarrow_\$ \mathrm{H}_0^A(\mathcal{A})] \leq 1/2 + \varepsilon_{\mathrm{H}} + T^2 \cdot (\varepsilon_{\mathrm{FSE}})$. $\square$

Theorem 15 (Theorem 5, restated). *Let* SK *be a* $(T, \varepsilon_{\mathrm{SK}})$*-secure symmetric-key encryption scheme and* $\mathcal{H}$ *be a* $(T, \varepsilon_{\mathcal{H}})$*-secure universal one-way hash function family. Then the* IRM *construction of Figure 8 instantiated with tree* τ *is correct,* $(T', h(\tau) \cdot \varepsilon_{\mathrm{SK}})$*-secure in the* $\mathrm{IND}_{\mathrm{IRM}}$ *game of Figure 7, and* $(T', h(\tau) \cdot (\varepsilon_{\mathrm{SK}} + T' \cdot \varepsilon_{\mathcal{H}}))$*-secure in the* $\mathrm{RIND}_{\mathrm{IRM}}$ *game of Figure 7, for* $T' \approx T$.

Proof. We will prove (more challenging) correctness and security of the robust scheme first. Correctness is clear since when any IRM.write(st_{sec}, C, i, d) is performed, $\tau.path(i)$ is replaced with a chain of encryptions that is used by IRM.read(st_{sec}, C, i) to, starting with root key $k_{\tau.r}$, decrypt d. Moreover, whenever any other IRM.write$(st_{sec}, C, i', \cdot)$ or IRM.del(st_{sec}, C, i') for $i' \neq i$ is performed before the next time IRM.write$(st_{sec}, C, i, \cdot)$ or IRM.del$(st_{sec}, C, i, \cdot)$ is performed, the encrypted keys/data at the nodes of children of the skeleton *skel* before the operation are still encrypted by the new keys at their parents in *skel* after the operation. Thus, IRM.read(st_{sec}, C, i) will still correctly return d. It is also clear that if the correct cells C are provided in all oracle queries by the adversary, then the hashes will always be accepted.

For an attacker $\mathcal{A}$ to win the security game, they must either

– Make some oracle query $\mathbf{Read}(i, \widetilde{C}, d)$, $\mathbf{Write}(i, \widetilde{C}, d)$, $\mathbf{Chall}(i, \widetilde{C}, d_0, d_1)$, or $\mathbf{Del}(i, \widetilde{C})$ with malicious cells $\widetilde{C} \neq C$, where $(\cdot, C) \leftarrow$ IRM.srvr-op(st_{pub}, i, op), that with non-negligible probability does not result in the challenger outputting $\perp$. Moreover, this query must cause the game to output **win** after some **Read**() query, or enable $\mathcal{A}$ to guess the challenge bit b correctly at the end of the game; or
– Provide the correct cells $\widetilde{C} = C$ for each such query, but still guess the challenge bit b correctly at the end of the game.

In the first case, we will reduce to the security of the underlying UOWHF family, $\mathcal{H}$. If $\mathcal{A}$ can provide malicious cells $\widetilde{C} \neq C$ without IRM outputting $\perp$, this means that $\mathcal{A}$ found an input to some hash h that is different than what IRM input to it, but still evaluates to the same value, thus breaking the UOWHF security.

More formally, we define reduction $\mathcal{R}$ attacking UOWHF family $\mathcal{H}$. $\mathcal{R}$ first guesses the query in which the hash corresponding to the node closest to the root that will eventually be broken by $\mathcal{A}$ when it provides $\widetilde{C} \neq C$ in some future query, is computed. It also guesses that node, v. It simulates IRM exactly up until that query. For that query, it simulates IRM exactly, except when computing the hash function and hash, (h_v, y_v) to be stored at v, it instead sends the computed input x to the hash to its challenger, and receives back (h_v, y_v). $\mathcal{R}$ continues simulating IRM exactly, until the query in which $\mathcal{A}$ provides incorrect $\widetilde{C} \neq C$. At this point, if $\mathcal{R}$ guessed correctly, it finds the input $x' \neq x$ to the hash h_v which evaluates to $h_v(x'_v) \rightarrow y_v$, and sends it to its challenger, winning the game. If it incorrectly guesses the query and node, $\mathcal{R}$ simply aborts. Thus, it is clear that $\Pr[\mathcal{R} \rightarrow_{\$} x' \neq x : h_v(x) = h_v(x')] = 1/T' \cdot 1/h(\tau) \cdot \Pr[\mathcal{A} \text{ wins } \mathrm{IND}_{\mathrm{IRM}}(\mathcal{A})]$.

In the second case, we will reduce to the security of the underlying symmetric key encryption scheme, SK. Note that in our scheme, every time IRM.write(st_{sec}, C, i, d) is performed, all of the keys on $\tau.path(i)$ (and $\tau.r$) are deleted or replaced. Additionally, in the security game, after a query to $\mathbf{Chall}(i, d_0, d_1)$, the adversary must again query $\mathbf{Write}(i, d)$ or $\mathbf{Del}(i)$ before querying $\mathbf{Expose}()$. Therefore, $\mathcal{A}$ never learns anything (beyond the ciphertexts) about the key $k_{i_{d(i)-1}}$ that directly encrypts d_b, nor keys k_{i_j} for $j \in [0, d(i) - 2]$, that encrypt $k_{i_{j+1}}$. So, from the security of SK, $\mathcal{A}$ cannot distinguish whether d_0 or d_1 was written to cell i.

More formally, let hybrid H_0 be the world in which IRM.write(st_{sec}, C, i, d_0) is performed when $\mathbf{Chall}(i, d_0, d_1)$ is queried, and hybrid H_j for $j \in [h(\tau) - 1]$ be defined exactly as H_{j-1}, except that $c_{i_j} \leftarrow_{\$}$ SK.enc($k_{i_{j-1}}, k_{i_j}$) in the algorithm is replaced with SK.enc($k_{i_{j-1}}, 0^\lambda$). Similarly, hybrid $\mathrm{H}_{h(\tau)}$ is defined exactly as $\mathrm{H}_{h(\tau)-1}$, except that $c_{i_{h(\tau)}} \leftarrow_{\$}$ SK.enc($k_{i_{j-1}}, d_0$) is replaced with SK.enc($k_{i_{j-1}}, d_1$).

We define reduction $\mathcal{R}_j$ for $j \in [h(\tau) - 1]$ attacking scheme SK to first initialize IRM.init($1^\lambda, n$) and output st_{pub} to $\mathcal{A}$. It then simulates all queries to

$\mathbf{Write}(i, d), \mathbf{Read}(i), \mathbf{Del}(i), \mathbf{Expose}()$ honestly. For the query to $\mathbf{Chall}(i, d_0, d_1)$, $\mathcal{R}_j$ executes as H_{j-1} does, except when creating c_{i_j}. Instead, $\mathcal{R}_j$ queries the oracle of the SK security game $\mathrm{SK.enc}(k_{i_{j-1}}, \cdot, \cdot)$ on $(0^\lambda, k'_{i_j})$. $\mathcal{R}_{h(\tau)}$ is defined similarly.

It is easy to see that for any $j \in [\log n]$, $\mathcal{R}_j$ simulates exactly H_{j-1} and H_j, except for when creating c_{i_j}. For c_{i_j}, depending on the challenge bit b' of the SK security game, $\mathcal{R}_j$ simulates $\mathrm{H}_{j-b'}$. Therefore, $\mathcal{A}$ cannot distinguish between H_{j-1} and H_j. A similar hybrid argument can be used to transform from $\mathrm{H}_{h(\tau)}$ to the world in which $\mathrm{IRM.write}(st_{sec}, st_{pub}, i, d_1)$ is performed.

Thus, IRM from Figure 8 is correct and $(T', h(\tau) \cdot (\varepsilon_{\mathrm{SK}} + T' \cdot \varepsilon_{\mathcal{H}}))$-secure in the $\mathrm{RIND}_{\mathrm{IRM}}$ game. It is easy to see its security in the $\mathrm{IND}_{\mathrm{IRM}}$ game by removing the first type of attacker and the reduction to the UOWHF family $\mathcal{H}$. $\square$

E Alternative via FS-KEM and AEAD

In Figure 9, we propose an alternative construction based on FS-KEM and AEAD (instead of FS-PKE and collision-resistant hash functions) that offers similar performance guarantees as our construction from Section 6.

```
$\underline{\mathrm{IKMR.gen}(1^\lambda)}$:

 1. set $t_0, t_1 \leftarrow 1$, $SK[\cdot], PK \leftarrow \bot$
 2. generate $(sk, pk) \leftarrow_\$$
      $\mathrm{FSK.gen}(1^\lambda, 1)$
 3. set $(SK[1], PK) \leftarrow$
      $((sk, t_0), pk)$
 4. return $((SK, t_0, t_1),$
      $(PK, t_0, t_1))$

$\underline{\mathrm{IKMR.re\text{-}gen}((SK, t_0, t_1))}$

 1. increment $t_1 \leftarrow t_1 + 1$
 2. generate $(sk, pk) \leftarrow_\$$
      $\mathrm{FSK.gen}(1^\lambda, t_1 - t_0 + 1)$
 3. set $(SK[t_1], PK) \leftarrow$
      $((sk, t_0), pk)$
 4. return $((SK, t_0, t_1),$
      $(PK, t_0, t_1))$

$\underline{\mathrm{IKMR.del}((SK, t_0, t_1), \ell)}$:

 1. for $i \in [\ell]$:
    (a) set $SK[t_0 + i - 1] \leftarrow \bot$
    (b) for $t \in [t_0 + \ell, t_1]$:
          set $(sk, t') \leftarrow SK[t]$;
          update $sk' \leftarrow \mathrm{FSK.up}(sk)$;
          store $SK[t] \leftarrow (sk', t')$
 2. increment $t_0 \leftarrow t_0 + \ell$
 3. return $(SK, t_0, t_1)$
```

```
$\underline{\mathrm{IKMR.enc}((PK, t_0, t_1))}$:

 1. sample random $k \leftarrow_\$ \mathcal{K}$
 2. compute $(k_{\mathrm{AE}}, c_{\mathrm{FSK}}) \leftarrow_\$$
      $\mathrm{FSK.enc}(PK, t_1 - t_0)$
 3. compute $c_{\mathrm{AE}} \leftarrow_\$$
      $\mathrm{AE.enc}(k_{\mathrm{AE}}, k, (c_{\mathrm{FSK}}, t_1))$
 4. return $(k, ((c_{\mathrm{FSK}}), c_{\mathrm{AE}}, (t_1)))$

$\underline{\mathrm{IKMR.re\text{-}enc}((PK, t_0, t_1), (C, c_{\mathrm{AE}}, T))}$:

 1. $(c_0^\circ, \ldots, c_l^\circ) \leftarrow C$; $(t_0^\circ, \ldots, t_l^\circ) \leftarrow T$
 2. if $t_0 > t_0^\circ$ return $\bot$
 3. compute $(k_{\mathrm{AE}}, c_1) \leftarrow_\$$
      $\mathrm{FSK.enc}(PK, t_0^\circ - t_0)$
 4. $C' \leftarrow (c_0^\circ, \ldots, c_l^\circ, c_1)$; $T' \leftarrow (t_0^\circ, \ldots, t_l^\circ, t_1)$
 5. compute $c_{\mathrm{AE}} \leftarrow_\$$
      $\mathrm{AE.enc}(k_{\mathrm{AE}}, c_{\mathrm{AE}}, (C', T'))$
 6. return $(C', c_{\mathrm{AE}}, T')$

$\underline{\mathrm{IKMR.dec}((SK, t_0, t_1), (C, c_{\mathrm{AE}}, T))}$:

 1. $(c_0^\circ, \ldots, c_l^\circ) \leftarrow C$; $(t_0^\circ, \ldots, t_l^\circ) \leftarrow T$
 2. $C_i \leftarrow (c_0^\circ, \ldots, c_i^\circ)$; $T_i \leftarrow (t_0^\circ, \ldots, t_i^\circ)$
 3. for $i$ from $l$ to 0:
    (a) let $(sk_i, t_0') \leftarrow SK[t_i^\circ]$; decrypt
          $k_{\mathrm{AE}} \leftarrow \mathrm{FSK.dec}(sk_i, t_0^\circ - t_0', c_i^\circ)$
          $c_{\mathrm{AE}} \leftarrow \mathrm{AE.dec}(k_{\mathrm{AE}}, c_{\mathrm{AE}}, (C_i, T_i))$
 4. set $k \leftarrow c_{\mathrm{AE}}$
 5. return $k$
```

Fig. 9. FS-KEM-based IKEM with Re-Encapsulations construction.