# Fair-Weather No More: Guaranteed Efficiency in Secure Group Messaging

James Bartusek[1], Nir Bitansky[2], Yevgeniy Dodis[2], Rachit Garg[2], and David J. Wu[3]

[1]Columbia University
[2]New York University
[3]UT Austin

**Abstract**

Secure group messaging protocols, now standardized by the IETF as Messaging Layer Security (MLS), provide end-to-end encryption for billions of users. The cryptographic core of these protocols is continuous group key agreement (CGKA), a primitive designed to maintain a shared secret among a dynamic group while providing security guarantees like forward secrecy and post-compromise security. A critical challenge for CGKA is achieving efficiency, particularly sublinear complexity (in the size of the group), for group operations. While practical tree-based protocols like TreeKEM offer logarithmic complexity in ideal (so-called "fair-weather") scenarios, their performance degrades to linear in the worst-case, and even realistic average-case, scenarios. This performance collapse raises the fundamental question of whether any CGKA protocol can achieve provably sublinear worst-case complexity.

Prior work has established significant barriers to this goal, including black-box impossibility results ruling out efficient constructions from standard public-key encryption. Theoretical solutions circumvent these barriers using powerful tools like indistinguishability obfuscation ($i\mathcal{O}$), but these constructions are astronomically inefficient and often provide weaker security guarantees, such as lacking forward secrecy. This leaves a wide gap between practical protocols with poor worst-case guarantees and theoretical solutions that are entirely impractical.

In this paper, we narrow this gap by presenting the first CGKA protocol that achieves provably *logarithmic worst-case complexity for both computation and communication.* Our first construction is based on a falsifiable and plausibly post-quantum assumption called decomposed learning with errors (decomposed LWE), and achieves basic CGKA security (only group members know the key) and post-compromise security, but *not* forward secrecy. We then show how to extend our scheme in the random oracle model to achieve optimal security (including forward secrecy) while retaining worst-case sublinear communication. However, the forward-secure refresh operation takes linear time in the group size, while still producing compact ciphertexts.

Our work is the first to establish that worst-case efficient CGKA is theoretically possible from simple falsifiable assumptions. Moreover, it offers a plausible roadmap towards concretely efficient constructions.

## 1 Introduction

Secure group messaging (SGM) is a cornerstone of modern digital communication, with billions of users relying on platforms like Signal, WhatsApp, and Facebook Messenger for end-to-end (E2E) encrypted conversations. The growing importance of SGM is underscored by the recent standardization effort by the IETF, which culminated in the Messaging Layer Security (MLS) protocol [BBR+23]. MLS is already seeing widespread adoption by major industry players such as Cisco, Discord, and Google.

The cryptographic core of SGM protocols (including the MLS standard) is a primitive known as *continuous group key agreement* (CGKA) [ACDT20]. CGKA provides a clean abstraction for maintaining a shared secret among a dynamic group of users, where members can be added, removed, or can perform state updates. CGKA could be

viewed as a functionality extension of *multicast encryption* [CGI+99, BDT22], which has a special group manager who is in charge of adding and removing users. In contrast, in a CGKA, there is no special group manager: any user can add or remove users to/from the group. The CGKA protocol at the heart of the MLS standard, known as *TreeKEM*, exemplifies this functionality. Although CGKA only considers a passive adversary, the composition theorems of [ACDT20, ACDT21] combine CGKA with symmetric-key primitives to yield much stronger levels of active attacker security for SGM. This is because CGKA still allows the passive adversary to corrupt the secret states of honest participants at any time. For now, we will consider the weakest, but already very challenging, security property of CGKA in light of such corruptions: the shared secret key is secure, provided all previously corrupted users are no longer in the group, and no current group member is corrupted from now on.[1]

A critical metric for the practicality of CGKA is its *communication efficiency*. Ideally, the initiator's communication cost of any group operation should be sublinear in the group size $n$, and preferably polylogarithmic. An even stronger requirement is sublinear *computational* complexity for each user, which implies sublinear communication while also disallowing inefficient individual operations that scale proportionally to $n$. Indeed, such logarithmic efficiency is easily achieved in existing multicast encryption schemes [CGI+99, BDT22], and we would like similar efficiency to hold in CGKA. At first glance, existing CGKA upper bounds seem promising. Tree-based protocols, like TreeKEM, follow the blueprint of multicast encryption schemes, and achieve an appealing $O(\log n)$ communication complexity by arranging users at the leaves of a binary tree. When a user updates, they refresh the keys on their path to the root, and encrypt these new keys for their siblings along the co-path, requiring only a logarithmic number of ciphertexts.

However, this efficiency is only guaranteed under "fair-weather" (or, really, *best-case*—see below) scenarios. For example, if a user who previously added many members is removed, all secrets they generated are now considered compromised to prevent "double-join" attacks [ACDT20, CCG+18], even if this (honest) user was supposed to erase such no-longer-needed secrets. This means that the remaining group members are often left with no shared secret structure with the users who were added by this user, forcing them to establish a new key "from scratch." In the worst case, this might require $\Omega(n)$ communication [BDG+22], which degrades to that of a trivial protocol (corresponding to a flat tree with $n$ leaves at level 1). In fact, the recent work of Auerbach et al. [ACEP25] showed that the communication complexity of TreeKEM quickly converges to being nearly linear in $n$ in the following simple scenario: suppose that every $d$ successive rounds, a random user $U_0$ sequentially announces key updates for a random collection of $d$ users $U_1, \ldots, U_d$. When $d = 1$, the work of [ACEP25] shows that the communication complexity is already $\Omega(\sqrt{n})$ and rises to $\Omega(n^{0.99})$ when $d = 50$.[2] One can reasonably interpret such a statement as saying that even the *average-case* complexity of TreeKEM is much closer to linear, and certainly nowhere near the desired $O(\log n)$ "fair-weather" complexity. In fact, a more recent experimental work of [SDFV+25] developed a highly configurable environment for empirical evaluations of MLS, and reached a similar conclusion experimentally: "our results show that computation costs scale linearly in practical scenarios even in the best-case scenario."

These results basically kill the fair-weather claim regarding MLS, and raise the following fundamental question:

*Do there exist CGKA protocols with sublinear worst-case complexity?*

This important question was recently studied, with several papers establishing some barriers to achieving this goal from standard cryptographic tools. These lower bounds come in several flavors:

- **Black-box impossibility:** Bienstock et al. [BDG+22] showed that it is impossible to build a worst-case efficient CGKA in a *black-box* way from public-key encryption (PKE). This means that any protocol that treats PKE as a sealed component—using it only for its intended encryption/decryption functionality without exploiting its internal mathematical structure—cannot avoid a linear $\Omega(n)$ communication cost in the worst case. This class of constructions includes TreeKEM and its variants.

- **Symbolic lower bounds:** Other works have proven lower bounds in more abstract, symbolic models, where cryptographic primitives are treated as ideal objects. In this idealized world, several papers [BDR20, AAB+24]

[1] This is the CGKA notion from [BDG+22], but without "forward" and "post-compromise" security properties that we will discuss later.

[2] Technically, the authors of [ACEP25] stated their result in the so-called "propose-then-commit" framework formally used by MLS, where users first propose changes, and some other user then "commits" to all of these changes at once. For simplicity of exposition, we state our results in the original CGKA framework of [ACDT20], where we do not separate proposals from commits (so every group change is a "commit" by default). This does not affect anything we say, but is easier to explain and analyze.

showed that achieving fast "post-compromise recovery" (a term we explain later, when considering stronger CGKA security) after $t$ *concurrent* user updates requires $\Omega(t \log(n/t))$ communication per such update.[3]

On the positive side, a completely different line of work has shown a path to circumventing these lower bounds using powerful primitives. In particular, Bienstock et al. [BDR20, BDG+22] observed a connection between CGKA and *multi-party non-interactive key exchange* (multi-party NIKE). In a multi-party NIKE protocol, a group of users can each publish a single message, and by reading all the public messages, everyone can compute a shared secret. This primitive can be used to realize a very basic form of CGKA where adding a user or updating the user's key only requires sending the new public key, and deleting just announces the identity of the removed user. In either case, the updated group roster simply recomputes a fresh multi-party NIKE key after the change. Such multi-party NIKE constructions, in turn, can be built from *indistinguishability obfuscation* ($i\mathcal{O}$) [BGI+01]—a "master tool" of modern cryptography that can make any program unintelligible while preserving its functionality. While there are some subtleties in the latter CGKA construction from $i\mathcal{O}$ [BZ14, KRS15], it was recently formalized by Alwen et al. [ABG+25], who constructed a *succinct* multi-party NIKE from $i\mathcal{O}$, meaning that the size of the public message broadcast by each member is independent of group size. However, this positive result, while achieving sublinear communication complexity, comes with significant limitations:

- **Impractical computational cost:** Current constructions of $i\mathcal{O}$ are astronomically inefficient, involving many layers of recursive composition of non-black-box primitives (like functional encryption), making them purely theoretical tools. In the words of the authors of [ABG+25], the resulting CGKA is therefore a "feasibility result, and not a proposal of a real-world scheme."
- **Lack of post-quantum security:** There are currently no $i\mathcal{O}$ constructions from well-studied assumptions that are post-quantum secure.
- **Communication-only focus:** The multi-party NIKE-based result, and another CGKA from a related primitive called *compact key exchange* (CKE) [BDG+22], primarily focus on minimizing the length of communication messages, without guarantees on the computational cost. Indeed, their constructions achieve linear $\Omega(n)$ computational complexity, as each (compact) group secret is recomputed from scratch. In contrast, in a good CGKA scheme, the entire *computational* complexity must be sublinear. This implies an important requirement we call *incrementality*. Given a compact state $\sigma_t$ of the user $U$ at time $t$, the state-key tuple $(\sigma_{t+1}, k_{t+1})$ at time $(t+1)$ should be computed in sublinear time from $\sigma_t$ and a single sublinear ciphertext $c_{t+1}$ broadcast by the group member making the corresponding change to the group.

The current state of affairs thus leaves a significant gap between theory and practice. On the one hand, we have practical protocols like TreeKEM that are efficient in limited (likely "best") situations, but provably inefficient in others (including realistic "average-case" ones). On the other hand, we have theoretical constructions that achieve the desired sublinearity, but are completely impractical and computation-heavy (even in theory). In contrast, the negative results, while powerful, only rule out specific classes of constructions, leaving open the possibility of a solution that is both practical and provably secure in the worst case.

To summarize, prior to this work, the status of worst-case efficient CGKA was unclear, casting doubt on whether it could ever be a truly ubiquitous and practical primitive for the real world. This brings us to our main question:

*Can we build a provably secure CGKA protocol with* worst-case *sublinear complexity which is potentially efficient for real-world implementation?*

In fact, motivated by our discussion, we would like to additionally ask the following sub-questions for which we would strongly prefer affirmative answers:

(a) Can the sublinear complexity be polylogarithmic in the group size $n$ and polynomial in the security parameter $\lambda$ (i.e., $\mathrm{poly}(\lambda, \log n)$)?

(b) Can the sublinear guarantee apply to the *overall computation* for each user, and not just the communication cost?

[3] Technically, these works were more aimed at addressing a different (unrelated) limitation of MLS, prohibiting concurrent commit operations (which is also the case in the notion of CGKA we study in this work). This was the real-world motivation to separate concurrent "proposals" from sequential "commits" in the MLS standard.

(c) Can we base such a construction on a falsifiable (and preferably, plausibly post-quantum) cryptographic assumption?

**Our main result.** In this paper, we provide a single construction that *simultaneously answers all of these questions in the affirmative*. We present the first CGKA protocol that is provably secure, achieves polylogarithmic worst-case complexity for both computation and communication, relies on standard assumptions, and is potentially concretely efficient enough for practical consideration (although the latter will likely require further optimization not pursued in this work). Our security model for CGKA is based on that of [BDG+22], in which the black-box lower bound mentioned above was shown (and in fact we also consider a strengthening that incorporates *forward secrecy*, as we discuss further below). Hence, for the first time, our work firmly establishes CGKA as a practical, real-world primitive backed by provable theoretical guarantees, even in the worst case.

At a high level, our scheme builds on finding a connection between CGKA and a primitive called *distributed broadcast encryption* (DBE) [WQZD10, BZ14], of which simple and relatively lightweight lattice-based constructions were recently obtained [CW24, CHW25, WW25]. In a DBE scheme, users can publish their initial public keys without any coordination. Given any group of $n$ users $U_1, \ldots, U_n$, any other user $U$ can send a compact ciphertext $c$. Every user $U_i$ can decrypt $c$ using their individual secret key together with knowledge of the $(n-1)$ public keys of the other users $U_j$. This is very similar to the compact key exchange notion suggested by [BDG+22], and almost immediately implies a simple CGKA scheme, where a group update message will simply encrypt a fresh group secret using the $n$ public keys corresponding to the current group roster after the update.

Similar to earlier constructions based on multi-party NIKE, however, the CGKA obtained in this way will not have the incrementality property that we seek. Thus, aside from establishing the connection between CGKA and distributed broadcast encryption, we will need to show how to make DBE schemes incremental. Indeed, this is precisely what we do. In Section 4, we formally define the notion of *incremental DBE*, and, in Section 6, we show that incremental DBE implies a CGKA scheme with comparable (sublinear) parameters. As one of our main technical results, in Section 4.1, we show how to build on the recent DBE scheme of Wee and Wu [WW25] to obtain an incremental distributed broadcast encryption scheme. We prove security in the plain model from the same falsifiable lattice assumptions used in [WW25] (i.e., the decomposed LWE assumption [AMR25]). This assumption is plausibly post-quantum secure. While the parameters for the current scheme are likely too large to be useful for real-world CGKA/SGM schemes, the scheme itself is implementable with current software/hardware. With further optimization, we believe our paper will pave the way to real-world SGM schemes with worst-case sublinear complexity.

**Towards stronger security.** Thus far, and similar to the prior work of Alwen et al. [ABG+25], we achieve the weakest security notion of CGKA, where we disallow the attacker from corrupting existing group members, even if this corruption happens far in the past or the future of the current key. In contrast, modern SGM schemes (and, correspondingly, their underlying CGKA schemes) strive for two stronger security goals: *post-compromise security* (PCS), which ensures that the group can recover a secure state after a compromise through normal protocol operations, and *forward secrecy* (FS), which guarantees that past group keys remain secure even if current states are leaked.

Fortunately, the PCS property is easily achieved by all CGKA schemes mentioned so far, including TreeKEM, the $i\mathcal{O}$ scheme from [ABG+25], or the new CGKA scheme we built from incremental distributed broadcast encryption.[4] Intuitively, the user with current identity $\mathsf{pk}_{\mathsf{old}}$ can conclude any group operation by sampling a new public key $\mathsf{pk}_{\mathsf{new}}$ for itself, removing its old public key $\mathsf{pk}_{\mathsf{old}}$ from the group, and re-inserting a "new user" $\mathsf{pk}_{\mathsf{new}}$ to the group. *Security-wise*, this achieves PCS, since the compromise of the old secret key $\mathsf{sk}_{\mathsf{old}}$ is no longer relevant by the basic CGKA security, as the user $\mathsf{pk}_{\mathsf{old}}$ is no longer in the group. *Efficiency-wise*, however, this could be problematic if the base CGKA scheme does not have sublinear add/remove operations, as was the case for all previous CGKA schemes. However, since our new DBE scheme is incremental, we achieve PCS for our CGKA without any efficiency loss.

Following [ACDT20], our definition also includes a per-user no-delete option, under which that user's past secret states are retained and exposed upon corruption. We establish security even when the adversary has this option, which rules out so-called "double-join" attacks. In such an attack, a user performing an add or remove may

[4] By adding the PCS requirement to our basic CGKA definition so far, we effectively arrive at the CGKA definition from [BDG+22], under which they proved their $\Omega(n)$ black-box impossibility result of building CGKA from public-key encryption. This is the model in which we actually prove the CGKA security of our new scheme, overcoming this black-box impossibility.

learn secret material associated with another user, and is later able to use this material to derive shared group keys, even after being removed itself. In order to prevent such attacks, the TreeKEM protocol requires worst-case linear communication, while our protocol prevents such attacks with only logarithmic communication.

Things are considerably more complicated with forward secrecy. For example, TreeKEM (and, thus, the current MLS standard) does not achieve good forward-secrecy properties, as was demonstrated by Alwen et al. [ACDT20]. The same trivially holds for schemes based on multi-party NIKE, such as [ABG+25]. In both cases, the issue comes from the fact that all these schemes lack the "ratcheting" mechanism needed to securely forget past information. In particular, corrupting a member of the evolving group who processed all updates, but never generated its *own* update, will clearly leak all the past group secrets, breaking forward secrecy. Going forward, we call the challenge of protecting the security of such users *updatability*, because their initial secrets have to be updated in the course of the group evolution, even if *they themselves do not post any updates*!

At a technical level, updatability is related to the notion of so-called *updatable* public-key encryption (UPKE) [JMM19, ACDT20], where the sender can help the receiver achieve forward secrecy by modifying the receiver's public key from $\mathsf{pk}_{\mathsf{old}}$ to $\mathsf{pk}_{\mathsf{new}}$ such that the following hold:

- the receiver can update its secret from $\mathsf{sk}_{\mathsf{old}}$ to $\mathsf{sk}_{\mathsf{new}}$, and then erase $\mathsf{sk}_{\mathsf{old}}$;
- compromising the new key $\mathsf{sk}_{\mathsf{new}}$ does not help to decrypt past ciphertexts for $\mathsf{pk}_{\mathsf{old}}$; and
- the "helping sender" cannot decrypt new ciphertexts for $\mathsf{pk}_{\mathsf{new}}$ without compromising the key $\mathsf{sk}_{\mathsf{old}}$, even if it knows the randomness used to go from $\mathsf{pk}_{\mathsf{old}}$ to $\mathsf{pk}_{\mathsf{new}}$.

Indeed, the only known CGKA scheme that achieves forward secrecy is RTreeKEM [ACDT20] and is based on a black-box use of an updatable public-key encryption scheme on top of TreeKEM. As such, it does not meet the efficiency requirements of this work. Thus, we can ask the following question:

*Can we build a provably secure CGKA protocol with* worst-case *sublinear complexity which is also forward secure?*

This question was wide open before our work, even if we allow heavy machinery, such as $i\mathcal{O}$. Intuitively, this appears quite hard, as building efficient updatable public-key encryption schemes [JMM19, ACDT20, HLP22, HPS23, AFM24, AFMR25] seems to require some type of algebraic structure, and it is unclear how to introduce such structure to a generic $i\mathcal{O}$-based construction like the ones described above.

**Our second result.** As a partial result to resolve this problem, in Section 5, we define and build an *updatable* (and still incremental) DBE scheme, which in turn gives the *first CGKA scheme with forward secrecy and worst-case sublinear communication complexity*. This scheme is noticeably more involved than our basic incremental DBE scheme, as it adds an updating mechanism to the incremental DBE. Namely, a sender implicitly updates the secret keys of *all* the receivers in the group (while satisfying the requirements listed above). Security in turn relies on the same falsifiable lattice assumptions as before, but we additionally need to work in the random oracle model.

Our new scheme resolves the main open problem above in the efficiency model explicitly considered in the prior works of [ABG+25, BDG+22], where communication is required to be compact, but computation is allowed to scale with the group size $n$. In fact, the *only* linear-time operation in our new CGKA scheme is a special "forward secrecy refreshing" operation, which results in a compact ciphertext and allows all group members to effectively "forget old secrets." This allows our scheme to achieve FS. The users can still add/remove other users in sublinear time, thereby preserving the standard CGKA security requirements.

In this scheme, each party needs to store the group roster and the public keys of the current members. Thus, the per-party storage is linear in the group size. This matches the standard model of the MLS and CGKA literature [ACDT20], where maintaining the roster is customary (required by MLS), and where the relevant efficiency measures are communication and computation in the RAM model; linear storage is therefore not considered a drawback.[5] We note that the efficiency model for our new scheme is still meaningful for situations where the network bandwidth is much more

[5] In contrast, our incremental DBE scheme from Section 4 has the property that each party in principle only needs to store its own aggregated key and not the complete set of public keys for all group members. However, doing so would require relaxing some security guarantees in the resulting CGKA and we do not explore this optimization here.

expensive than local computation, or where the users choose to run the refreshing operation infrequently (settling for a suboptimal notion of forward secrecy). For large groups, our scheme is also considerably more efficient than RTreeKEM (the only previous CGKA scheme with forward secrecy), since we achieve *worst-case* sublinear communication.

We leave it as an open problem to achieve a truly (computation-wise!) sublinear CGKA scheme with forward secrecy. As a partial indication of why our approach does not resolve this problem, in Appendix A, we show a simple attack on an aggressive, *sublinear-time variant* of our updatable DBE scheme from Section 5, which completely breaks the scheme.

# 2 Technical Overview

We start from the syntax of a distributed broadcast encryption scheme [WQZD10, BZ14] and then discuss the new properties that are essential for our application.

**Distributed broadcast encryption.** In a distributed broadcast encryption scheme [WQZD10, BZ14], a system administrator starts by running a Setup algorithm to generate a set of public parameters pp.[6] Each user is associated with an index $i \in \mathbb{N}$ and can independently sample a public/private key-pair $(\mathsf{pk}_i, \mathsf{sk}_i)$. Anyone can encrypt a message $m$ to a set of public keys $\{(i, \mathsf{pk}_i)\}_{i \in S}$, provided that each public key $\mathsf{pk}_i$ is associated with a distinct index. Any user $j \in S$ can decrypt ct using their secret key $\mathsf{sk}_j$ (together with knowledge of the other users' public keys). Specifically, the syntax can be summarized as follows:

$$\mathsf{ct} \leftarrow \mathsf{Enc}\left(\mathsf{pp}, \{(i, \mathsf{pk}_i)\}_{i \in S}, m\right), \qquad \mathsf{Dec}\left(\mathsf{pp}, \{(i, \mathsf{pk}_i)\}_{i \in S}, (j, \mathsf{sk}_j), \mathsf{ct}\right) = m.$$

The security requirement is that the ciphertext should computationally hide the message if one does not possess a secret key $\mathsf{sk}_i$ for any index $i \in S$. The efficiency requirement states that ciphertexts should be succinct: ideally, $|\mathsf{ct}| = \mathsf{poly}(\lambda, \log |S|)$.

**Incrementality and updatability.** We now discuss two new properties that are important for our application to efficient continuous group key-agreement (CGKA): *incrementality* and *updatability*.

- **Incrementality:** The incrementality property provides an efficient mechanism to update encryption keys as users join or leave the system. This property will be essential for supporting sublinear-time group updates in CGKA.
- **Updatability:** The updatability property establishes a key-rotation mechanism that will be helpful for achieving forward secrecy with sublinear communication.

**Incremental DBE.** An incremental distributed broadcast encryption scheme has similar Setup and KeyGen algorithms to vanilla DBE. In addition, in this setting, a group of users (associated with indices $i$ and public keys $\mathsf{pk}_i$) can compute an aggregated public key apk that represents a "group public key." We require the aggregate public key to be short (sublinear, and ideally, polylogarithmic, in the size of the group). In addition, each user also has their own aggregated secret key $\mathsf{ask}_i$, which is a function of their secret key $\mathsf{sk}_i$ and the public keys of the other users in the group. The aggregated public key is used to encrypt to the group and each user $i$ uses their aggregated secret key to decrypt:

$$\mathsf{ct} \leftarrow \mathsf{Enc}\,(\mathsf{apk}, m) \qquad \mathsf{Dec}(\mathsf{ask}_i, \mathsf{ct}) = m.$$

This syntax can be viewed as a registered version of broadcast encryption.[7] The key requirement in incremental DBE is the ability for group members to efficiently *update* their aggregated public and secret keys whenever a user joins

[6] Some schemes [BZ14, FWW23, WW25], including ours, either do not require a setup or support a *transparent* setup where the public parameters can be described by a uniform random string. Schemes based on bilinear maps [KMW23, GLWW23, GKPW24, WW26] require a structured setup.

[7] In registration-based cryptography [GHMR18], users independently choose their own public/secret keys and then there is a deterministic aggregation procedure that aggregates the public keys together into a short master public key. In the case of registration-based encryption, the short public key functions as the public key for an identity-based encryption scheme. In our setting, the aggregated key functions as the public key for a broadcast encryption scheme.

or leaves the group. Specifically, we define a pair of update algorithms IncSK, IncPK with the following syntax:

$$\mathsf{apk}' \leftarrow \mathsf{IncPK}\left(\mathsf{pp}, \mathsf{apk}, (j, \mathsf{pk}_j), \mathsf{op}\right) \qquad \mathsf{ask}_i' \leftarrow \mathsf{IncSK}\left(\mathsf{pp}, \mathsf{ask}_i, (j, \mathsf{pk}_j), \mathsf{op}\right),$$

where $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$. When a user joins (resp., leaves) the system, the remaining users in the group run the IncPK algorithm with the public parameters pp, the current group public key apk, the public key $\mathsf{pk}_j$ and index $j$ of the user joining (resp., leaving) the group and $\mathsf{op} = \mathsf{Add}$ (resp., $\mathsf{op} = \mathsf{Remove}$) to obtain the updated public key for the group. Similarly, they run IncSK with the public parameters pp, their current secret key $\mathsf{ask}_i$, the public key $\mathsf{pk}_j$ and index $j$ of the user joining (resp., leaving) the group, and $\mathsf{op} = \mathsf{Add}$ (resp., $\mathsf{op} = \mathsf{Remove}$) to obtain their new aggregated secret key. Finally, we require that the algorithms IncPK, IncSK run in time $\mathsf{poly}(\lambda)$, independent of the size of the group.

**Updatable DBE.** Incrementality alone does not provide forward secrecy. If a secret key for a user is ever compromised, the adversary gains the ability to decrypt all ciphertexts encrypted to the group's public key prior to the compromise. To address this, we augment the DBE scheme with an additional *refresh* mechanism. We refer to the resulting scheme as an updatable DBE scheme. An updatable DBE scheme consists of two additional algorithms UpdPK, UpdSK that users can use to refresh the aggregated public key and their aggregated secret key. In the group messaging application, these algorithms would be run whenever a user sends or receives a message. Formally, the update algorithms have the following syntax:

$$(\mathsf{ctref}, \mathsf{apk}', S') \leftarrow \mathsf{UpdPK}\left(\mathsf{pp}, \mathsf{apk}, S\right), \qquad (\mathsf{sk}_i', \mathsf{ask}_i', S') \leftarrow \mathsf{UpdSK}\left(\mathsf{pp}, \mathsf{sk}_i, \mathsf{ask}_i, \mathsf{ctref}, S\right).$$

In this case, the UpdPK algorithm takes the public parameters pp, the current group public key apk, and the set $S$ of user indices (that are group members), and outputs an updated public key $\mathsf{apk}'$, the refreshed set $S'$ of member public keys, and some additional metadata ctref that users can use to update their aggregated secret keys. In turn, users can use the UpdSK algorithm together with the public parameters pp, their secret key $\mathsf{sk}_i$, their current aggregated secret key $\mathsf{ask}_i$, the metadata ctref, and the current set $S$ of user indices to compute an updated secret key $\mathsf{sk}_i'$, an updated aggregated secret key $\mathsf{ask}_i'$, and the refreshed set $S'$ of member public keys (which contains their updated public key $\mathsf{pk}_i' = S'[i]$).

The UpdPK and UpdSK algorithms enable a key-rotation system that each user can use to update their public and secret keys on each communication. Our main efficiency requirement is that the outputs of UpdPK, UpdSK have size at most $\mathsf{poly}(\lambda)$, independent of the size of the group.[8] The output size of these update algorithms directly translates to the communication complexity of the messaging application. In the setting of public-key encryption, there is a rich line of work starting from constructions based on hierarchical identity-based encryption (which were rather involved) and the more natural relaxation called updatable public-key encryption [JMM19, ACDT20]. In our constructions, the main challenge in updatability is ensuring that the size of the updates, and thus the communication, remains sublinear in the size of the group $S$.

If we naïvely use a distributed broadcast encryption scheme as a black box to obtain an incremental (or updatable) DBE, it is challenging to achieve the required efficiency properties. This is because the syntax of distributed broadcast encryption does not natively support pre-aggregating public keys together. Instead, the public keys of the users are always provided as inputs to the encryption and decryption algorithms, so even if we are encrypting to the *same* group multiple times, the running time of the encryption and decryption algorithms is always linear in the size of the group. As it turns out though, most of the distributed broadcast encryption schemes we are aware of support the ability to first aggregate the public keys for many users into a short public key for the group [KMW23, GLWW23, CW24, GKPW24, CHW25, WW25, WW26]. In this work, we will use the recent lattice-based DBE scheme from [WW25] as the starting point for constructing our incremental and updatable DBE schemes.

**Background: matrix commitments.** The key building block underlying the distributed broadcast encryption scheme from [WW25] is the matrix commitment introduced by Wee [Wee25]. At a high level, Wee's matrix commitment scheme allows one to commit to a sparse matrix $\mathbf{M} \in \mathbb{Z}_q^{n \times L}$ (e.g., we can think of $L = 2^\lambda$ as being exponential,

[8] Note that here, we can also ask for an even stronger efficiency requirement where the algorithms do not need to take the current set of users $S$ as input and thus can run in time $\mathsf{poly}(\lambda)$. It is an interesting challenge to construct a scheme with this level of efficiency. This seems challenging for the scheme we put forward in this paper, and indeed, in Appendix A, we describe a simple attack on a natural variant of our scheme that would satisfy this stronger efficiency requirement.

but $\mathbf{M}$ only has a polynomial number of non-zero columns, and thus, a polynomial-size description). Specifically, given a set of public parameters $\mathsf{pp}_{\mathsf{com}}$ and a matrix $\mathbf{M}$, there is an efficient *deterministic* algorithm that outputs a commitment $\mathbf{C} \in \mathbb{Z}_q^{n \times m}$ together with low-norm "local openings" $\mathbf{z}_i \in \mathbb{Z}_q^{\hat{m}}$ such that

$$\mathbf{C}\mathbf{v}_i = \mathbf{m}_i - \mathbf{B}\mathbf{z}_i. \tag{2.1}$$

Here $\mathbf{B} \in \mathbb{Z}_q^{n \times \hat{m}}$ is a matrix and $\mathbf{v}_i \in \mathbb{Z}_q^m$ is a low-norm vector determined by the public parameters $\mathsf{pp}_{\mathsf{com}}$, and $\mathbf{m}_i$ denotes the $i^{\text{th}}$ column of $\mathbf{M}$. Importantly, the commitment $\mathbf{C}$ is compact: its dimensions are *independent* of the length of the original matrix $\mathbf{M}$ (i.e., $m = O(n \log q)$). Here, $\hat{m} = 4m$[5]. Since the commitment algorithm is deterministic, anyone can compute $\mathbf{C}$ and the openings $\mathbf{z}_i$ given the public parameters $\mathsf{pp}_{\mathsf{com}}$ and the matrix $\mathbf{M}$.

**Background: distributed broadcast encryption.** We next sketch a variant of the distributed broadcast encryption scheme from [WW25]. We consider a scheme that can support up to $L = 2^\lambda$ users. In the following description, we will write $\underset{\sim}{\mathbf{s}^\intercal \mathbf{A}}$ to denote $\mathbf{s}^\intercal \mathbf{A} + \mathbf{e}^\intercal$ where $\mathbf{e}$ is a low-norm vector.

- **Public parameters:** Sample the public parameters for the matrix commitment scheme $\mathsf{pp}_{\mathsf{com}}$, together with matrices $\mathbf{A} \in \mathbb{Z}_q^{n \times m}, \mathbf{p} \in \mathbb{Z}_q^n$. Recall that the matrix $\mathbf{B} \in \mathbb{Z}_q^{n \times \hat{m}}$ is defined by the public parameters for the matrix commitment scheme. Here, $(\mathbf{B}, \mathbf{p})$ can be viewed as the public key for a dual-Regev encryption scheme [Reg05], and $\mathbf{A}$ is used to re-randomize the ciphertexts.
- **User key generation:** Sample $\mathbf{r}_i \xleftarrow{\text{R}} \{0,1\}^{\hat{m}}$. Set the public key and secret key as

  $$\mathsf{pk}_i = \mathbf{t}_i \;=\; \mathbf{B}\mathbf{r}_i + \mathbf{p} - \mathbf{A}\mathbf{v}_i \qquad \text{and} \qquad \mathsf{sk}_i = \mathbf{r}_i.$$

  Here $\mathbf{v}_i \in \mathbb{Z}_q^m$ is the low-norm vector determined by the public parameters $\mathsf{pp}_{\mathsf{com}}$.
- **Encryption:** Given the set of recipients $\{(i, \mathbf{t}_i)\}_{i \in S}$ and a message $\mu \in \{0,1\}$, the encryption algorithm proceeds as follows:
  - For each $i \in S$, compute a matrix commitment $\mathbf{C}_i$ to the matrix with a single non-zero column $\mathbf{u}_i^\intercal \otimes \mathbf{t}_i$ where $\mathbf{u}_i \in \{0,1\}^L$ is the $i^{\text{th}}$ standard basis vector. Namely, the $i^{\text{th}}$ column of $\mathbf{C}_i$ is $\mathbf{t}_i$ and it is 0 elsewhere.
  - For any $i^* \in S$, let $\mathbf{z}_{i,i^*}$ be the local opening of $\mathbf{u}_i^\intercal \otimes \mathbf{t}_i$, such that Eq. (2.1) is satisfied. This means

    $$\mathbf{C}_i \mathbf{v}_{i^*} = \begin{cases} \mathbf{t}_i - \mathbf{B}\mathbf{z}_{i,i^*} & i = i^* \\ -\mathbf{B}\mathbf{z}_{i,i^*} & i \neq i^*, \end{cases}$$

    where $\mathbf{v}_{i^*}, \mathbf{z}_{i,i^*}$ are low-norm vectors.
  - Sample an LWE secret $\mathbf{s} \xleftarrow{\text{R}} \mathbb{Z}_q^n$ and, writing $\xi = \lfloor q/2 \rfloor$ for the scaling factor, output the ciphertext

    $$\mathsf{ct} \;=\; \left( \underset{\sim}{\mathbf{s}^\intercal \mathbf{B}},\; \underset{\sim}{\mathbf{s}^\intercal \left(\mathbf{A} + \textstyle\sum_{i \in S} \mathbf{C}_i\right)},\; \underset{\sim}{\mathbf{s}^\intercal \mathbf{p} + \mu \cdot \xi} \right). \tag{2.2}$$

    We note here that the DBE scheme from [WW25] defined $\mathbf{C}$ to be the commitment to $\sum_{i \in S} \mathbf{u}_i^\intercal \otimes \mathbf{t}_i$. In our variant, we take $\mathbf{C} = \sum_{i \in S} \mathbf{C}_i$ where each $\mathbf{C}_i$ is a commitment to $\mathbf{u}_i^\intercal \otimes \mathbf{t}_i$. The approach from [WW25] has the advantage that it can be instantiated with smaller parameters, whereas the variant we describe here directly enables incrementality (without requiring group members to remember the public keys of other users).
- **Decryption:** Take any index $i^* \in S$. Then, $\mathbf{v}_{i^*}$ is a public vector such that

  $$\underset{\sim}{\mathbf{s}^\intercal \left(\mathbf{A} + \sum_{i \in S} \mathbf{C}_i\right) \mathbf{v}_{i^*}} \;\approx\; \mathbf{s}^\intercal \mathbf{A}\mathbf{v}_{i^*} + \sum_{i \in S} \mathbf{s}^\intercal \mathbf{C}_i \mathbf{v}_{i^*} \;=\; \mathbf{s}^\intercal \mathbf{A}\mathbf{v}_{i^*} + \mathbf{s}^\intercal \mathbf{t}_{i^*} - \sum_{i \in S} \mathbf{s}^\intercal \mathbf{B}\mathbf{z}_{i,i^*}.$$

  Given $\{(i, \mathbf{t}_i)\}_{i \in S}$, the decrypter can compute each $\mathbf{z}_{i,i^*}$ and compute $\sum_{i \in S} \mathbf{z}_{i,i^*}$. Using the fact that $\mathbf{t}_{i^*} = \mathbf{B}\mathbf{r}_{i^*} + \mathbf{p} - \mathbf{A}\mathbf{v}_{i^*}$ together with the secret $\mathbf{r}_{i^*}$, the decrypter computes

  $$\underset{\sim}{\mathbf{s}^\intercal \left(\mathbf{A} + \textstyle\sum_{i \in S} \mathbf{C}_i\right) \mathbf{v}_{i^*}} \;-\; \underset{\sim}{\mathbf{s}^\intercal \mathbf{B}\, \mathbf{r}_{i^*}} \;-\; \underset{\sim}{\mathbf{s}^\intercal \mathbf{B} \textstyle\sum_{i \in S} \mathbf{z}_{i,i^*}} \;\approx\; \mathbf{s}^\intercal \mathbf{p}. \tag{2.3}$$

  The decrypter then subtracts this value from the third ciphertext component $\mathbf{s}^\intercal \mathbf{p} + \mu \cdot \xi$ and rounds to recover $\mu$.

From the properties of the matrix commitment scheme, $|\mathsf{pp}|, |\mathsf{pk}|, |\mathsf{sk}|, |\mathsf{ct}| = \mathsf{poly}(\lambda)$ and thus the distributed broadcast encryption scheme is efficient. Security in turn follows from the decomposed learning with errors (decomposed LWE) assumption [AMR25]. Specifically, under the decomposed LWE assumption, Wee [Wee25] showed that the following distributions are computationally indistinguishable

$$(\mathsf{pp}_{\mathsf{com}}, \underbrace{\mathbf{s}^\intercal \mathbf{B}}) \quad \text{and} \quad (\mathsf{pp}_{\mathsf{com}}, \mathbf{v}^\intercal),$$

where $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$ and $\mathbf{v} \xleftarrow{R} \mathbb{Z}_q^{\hat{m}}$. As shown in [WW25], this is sufficient to prove (selective) security of the above distributed broadcast encryption scheme. In the selective security game, the adversary must declare its challenge set ahead of time. In this case, the reduction algorithm can sample the public keys for the honest users $\mathsf{pk}_i = \mathbf{t}_i$ at setup time, pre-compute the commitments $\mathbf{C}_i$ to $\mathbf{u}_i^\intercal \otimes \mathbf{t}_i$, and then simulate the public parameters as $\mathbf{A} = \mathbf{BD} - \sum_{i \in S} \mathbf{C}_i$ and $\mathbf{p} = \mathbf{Bd}$, where $\mathbf{D}$ and $\mathbf{d}$ are low-norm matrices/vectors chosen by the reduction. In this case, the challenge ciphertext has the structure

$$\mathsf{ct} = \left(\underbrace{\mathbf{s}^\intercal \mathbf{B}}\,,\ \underbrace{\mathbf{s}^\intercal \left(\mathbf{A} + \textstyle\sum_{i \in S} \mathbf{C}_i\right)}\,,\ \underbrace{\mathbf{s}^\intercal \mathbf{p} + \mu \cdot \xi}\right) = \left(\underbrace{\mathbf{s}^\intercal \mathbf{B}}\,,\ \underbrace{\mathbf{s}^\intercal \mathbf{BD}}\,,\ \underbrace{\mathbf{s}^\intercal \mathbf{Bd} + \mu \cdot \xi}\right),$$

which is pseudorandom under the decomposed LWE assumption.

**Incrementality.** We start by observing that the DBE scheme described above already supports natural public-key aggregation and secret-key aggregation algorithms. Namely, given a group of users $S$, we can define the aggregated public key to be

$$\mathsf{apk} = \sum_{i \in S} \mathbf{C}_i \in \mathbb{Z}_q^{n \times m}.$$

Observe that given $\mathsf{apk}$, we can construct the ciphertext (Eq. (2.2)) in time that is independent of the number of users in the group. Similarly, each user $i^* \in S$ can pre-compute an aggregated decryption key $\mathsf{ask}_{i^*}$ as follows:

$$\mathsf{ask}_{i^*} \ = \ \left(\mathbf{v}_{i^*}, \mathbf{r}_{i^*}, \sum_{i \in S} \mathbf{z}_{i,i^*}\right),$$

where $\mathbf{z}_{i,i^*}$ is the local opening of $\mathsf{pk}_i = \mathbf{u}_i^\intercal \otimes \mathbf{t}_i$ at index $i^*$. Given the aggregated secret key $\mathsf{ask}_{i^*}$, evaluating the decryption relation (Eq. (2.3)) completes in time that is independent of the size of the set. Since the aggregated public key and the aggregated secret key for each user can be expressed as a linear function of the (committed) public keys of the users in the group and their respective openings, we obtain a simple aggregation mechanism. Specifically, we define $\mathsf{IncPK}, \mathsf{IncSK}$ as follows:

- To add a new user with index $j$ and public key $\mathsf{pk}_j = \mathbf{t}_j$ to the group, the $\mathsf{IncPK}$ algorithm first computes the commitment $\mathbf{C}_j$ to $\mathbf{t}_j$ and updates $\mathsf{apk} = \mathsf{apk} + \mathbf{C}_j$. Similarly, the $\mathsf{IncSK}$ algorithm for each user $i^* \in S$ would compute $\mathbf{z}_{j,i^*}$ as the local opening of $\mathbf{u}_j^\intercal \otimes \mathbf{t}_j$ with respect to position $i^*$ and accumulate $\mathbf{z}_{j,i^*}$ into the third component of $\mathsf{ask}_{i^*}$.
- To remove a user with index $j$ and public key $\mathsf{pk}_j = \mathbf{t}_j$, we simply perform the reverse operation. The new public key becomes $\mathsf{apk} = \mathsf{apk} - \mathbf{C}_j$ and each user would subtract $\mathbf{z}_{j,i^*}$ from the third component of $\mathsf{ask}_{i^*}$.

From the efficiency properties of the matrix commitment scheme, we have that adding and removing users has strictly sublinear update complexity.

**Updatability.** Next, we turn our attention to *updatability* and forward security. If an adversary compromises a user's secret key at time $t$, then all prior ciphertexts encrypted under the same aggregated public key become decryptable. To address this, we equip our scheme with an explicit *refresh* mechanism that re-randomizes all of the users' secret keys while retaining correctness. Moreover, this refresh mechanism preserves communication succinctness: the size of the update message communicated to the group is polynomial in the security parameter and independent of the size of the set $S$.

**Warm-up: public-key refresh in vanilla dual-Regev encryption.** Our approach follows a strategy similar to that taken in the group-based setting [JMM19, ACDT20]. We start by illustrating our technique for the simpler case of a vanilla dual-Regev encryption scheme [Reg05, GPV08] with public key $(\mathbf{B}, \mathbf{t})$. A user's secret key is a low-norm vector $\mathbf{r} \in \mathbb{Z}_q^{\hat{m}}$ such that $\mathbf{Br} = \mathbf{t} \in \mathbb{Z}_q^n$. In this case, a natural refresh strategy is to sample a random low-norm offset $\boldsymbol{\delta} \xleftarrow{R} \{0,1\}^{\hat{m}}$ and update the secret key to be $\mathbf{r} = \mathbf{r} + \boldsymbol{\delta}$. The corresponding public key for the user then becomes $\mathbf{t} = \mathbf{t} + \mathbf{B}\boldsymbol{\delta}$. Whenever the sender is sending a message to the receiver, they can now include an encryption of $\boldsymbol{\delta}$ with the message. The receiver can recover $\boldsymbol{\delta}$ and then update their state accordingly.

However, proving security of this toy scheme is nontrivial. After a compromise, the adversary learns the updated secret key $\mathbf{r} + \boldsymbol{\delta}$ along with an encryption of $\boldsymbol{\delta}$ (under LWE). Our goal is to show that even given the updated secret $\mathbf{r} + \boldsymbol{\delta}$ and an encryption of $\boldsymbol{\delta}$, ciphertexts encrypted with respect to the original public key $\mathbf{t}$ remain semantically secure. This introduces a circular dependency: the adversary sees $\mathbf{r} + \boldsymbol{\delta}$ (i.e., a masked version of the secret key $\mathbf{r}$) along with an encryption of $\boldsymbol{\delta}$. Prior works [JMM19, ACDT20, DKW21] address this circularity either by working in the random oracle model or by leveraging key-dependent-message security (KDM-security) of LWE against linear functions. In this work, we adopt a technique based on the random oracle heuristic. Concretely, in our construction, instead of shifting their key by $\boldsymbol{\delta} \in \{0,1\}^{\hat{m}}$, the user computes $\boldsymbol{\Delta} = H(\boldsymbol{\delta}) \in \{0,1\}^{\hat{m}}$ where $\boldsymbol{\delta} \in \{0,1\}^{\lambda}$ and the hash function $H$ is now modeled as a random oracle. The user then updates their public key $\mathbf{t}$ and secret key $\mathbf{r}$ as follows:

$$\mathbf{t} = \mathbf{t} + \mathbf{B}\boldsymbol{\Delta}, \qquad \mathbf{r} = \mathbf{r} + \boldsymbol{\Delta}.$$

As before, the sender would include an encryption of $\boldsymbol{\delta}$ with the ciphertext. The receiver can derive $\boldsymbol{\Delta}$ from $\boldsymbol{\delta}$ and perform the key update. In a sense, the random oracle is used here to "sever" the correlation between the blinded key $\mathbf{r} + \boldsymbol{\Delta}$ and the encryption of the blinding factor $\boldsymbol{\delta}$. The security argument roughly proceeds as follows:

- Let the public key be $\mathsf{pk} = (\mathbf{B}, \mathbf{t})$ and let $\mathbf{r}$ be the associated secret key. In the real distribution, the adversary sees pk, the refreshed secret key $\mathbf{r} + \boldsymbol{\Delta}$, and an encryption of $\boldsymbol{\delta} \in \{0,1\}^{\lambda}$ under pk. In dual Regev encryption, the ciphertext is the pair $(\underbrace{\mathbf{SB}}, \underbrace{\mathbf{St} + \boldsymbol{\delta} \cdot \xi})$, where $\mathbf{S} \xleftarrow{R} \mathbb{Z}_q^{\lambda \times n}$ and $\xi = \lfloor q/2 \rfloor$ is a scaling factor. The adversary's view can thus be described by the tuple

$$\left(\mathbf{B}, \mathbf{B}\boldsymbol{\Delta}, \mathbf{t}, \mathbf{r} + \boldsymbol{\Delta}, \underbrace{\mathbf{SB}}, \underbrace{\mathbf{St} + \boldsymbol{\delta} \cdot \xi}\right) = \left(\mathbf{B}, \mathbf{B}\boldsymbol{\Delta}, \mathbf{Br}, \mathbf{r} + \boldsymbol{\Delta}, \mathbf{D}, \underbrace{\mathbf{SB}}, \underbrace{\mathbf{SBr} + \boldsymbol{\delta} \cdot \xi}\right).$$

- Suppose the original secret $\mathbf{r}$ is drawn from a sufficiently wide distribution (e.g., $\mathbf{r} \xleftarrow{R} [B]^{\hat{m}}$ where $B$ is super-polynomial). Then, by a smudging argument, for all $\boldsymbol{\Delta} \in \{0,1\}^{\hat{m}}$, the distribution of $\mathbf{r} + \boldsymbol{\Delta}$ where $\mathbf{r} \xleftarrow{R} [B]^{\hat{m}}$ is statistically close to that of $\mathbf{r}' \xleftarrow{R} [B]^{\hat{m}}$. Reparametrizing $\mathbf{r} = \mathbf{r}' - \boldsymbol{\Delta}$, the adversary's view is statistically indistinguishable from

$$\left(\mathbf{B}, \mathbf{B}\boldsymbol{\Delta}, \mathbf{B}(\mathbf{r}' - \boldsymbol{\Delta}), \mathbf{r}', \underbrace{\mathbf{SB}}, \underbrace{\mathbf{SB}(\mathbf{r}' - \boldsymbol{\Delta}) + \boldsymbol{\delta} \cdot \xi}\right). \tag{2.4}$$

  Note that after this step the revealed secret $\mathbf{r}'$ no longer depends on $\boldsymbol{\Delta}$.

- Recall that $\boldsymbol{\Delta} = H(\boldsymbol{\delta}) \in \{0,1\}^{\hat{m}}$, with $H$ modeled as a random oracle. Suppose the adversary does not query $H$ on $\boldsymbol{\delta}$. Then $\boldsymbol{\Delta}$ is uniform over $\{0,1\}^{\hat{m}}$, and by the leftover hash lemma applied to the matrix $\mathbf{B}$, the following distributions are statistically indistinguishable:

$$(\mathbf{B}, \mathbf{B}\boldsymbol{\Delta}) \quad \text{and} \quad (\mathbf{B}, \mathbf{u}_1),$$

  where $\mathbf{u}_1 \xleftarrow{R} \mathbb{Z}_q^n$. Writing $\mathbf{B}(\mathbf{r}' - \boldsymbol{\Delta}) = \mathbf{Br}' - \mathbf{B}\boldsymbol{\Delta} = \mathbf{Br}' - \mathbf{u}_1$, the adversary's view in this case is statistically indistinguishable from

$$\left(\mathbf{B}, \mathbf{u}_1, \mathbf{Br}' - \mathbf{u}_1, \mathbf{r}', \underbrace{\mathbf{SB}}, \underbrace{\mathbf{S}(\mathbf{Br}' - \mathbf{u}_1) + \boldsymbol{\delta} \cdot \xi}\right).$$

- Now, by the LWE assumption, the following distributions are computationally indistinguishable:

$$\left([\mathbf{B} \mid \mathbf{u}_1], \underbrace{\mathbf{S}[\mathbf{B} \mid \mathbf{u}_1]}\right) \quad \text{and} \quad \left([\mathbf{B} \mid \mathbf{u}_1], [\mathbf{V} \mid \mathbf{u}_2]\right),$$

where $\mathbf{V} \xleftarrow{R} \mathbb{Z}_q^{\lambda \times \hat{m}}$ and $\mathbf{u}_2 \xleftarrow{R} \mathbb{Z}_q^{\lambda}$. Correspondingly, this means the adversary's view is computationally indistinguishable from

$$(\mathbf{B}, \mathbf{u}_1, \mathbf{B}\mathbf{r}' - \mathbf{u}_1, \mathbf{r}', \mathbf{V}, \mathbf{V}\mathbf{r}' - \mathbf{u}_2 + \boldsymbol{\delta} \cdot \xi) \,. \tag{2.5}$$

Since $\mathbf{u}_2$ is uniform and independent of everything else, the shift $\boldsymbol{\delta}$ is information-theoretically hidden. (The same argument shows that any message encrypted under pk is also hidden.)

- The above sequence of hybrids relies on the assumption that the adversary does not query the random oracle on $\boldsymbol{\delta}$. This is to ensure that $\boldsymbol{\Delta}$ is information-theoretically hidden from the view of the adversary. To argue this, suppose that the adversary makes at most $Q$ random oracle queries. We define $Q+1$ intermediate hybrid experiments $\mathsf{Hyb}_0, \ldots, \mathsf{Hyb}_Q$ where the adversary's view in each experiment is distributed according to Eq. (2.4). The output in $\mathsf{Hyb}_i$ is 1 if the adversary queries the random oracle on $\boldsymbol{\delta}$ as one of its first $i$ queries. Our goal is to show that the probability that the output in $\mathsf{Hyb}_Q$ is 1 is negligible. In this case, the above sequence of hybrid experiments shows that the distribution in Eq. (2.4) is computationally indistinguishable from the distribution in Eq. (2.5). We consider each pair of adjacent experiments. By construction, the outputs in $\mathsf{Hyb}_i$ and $\mathsf{Hyb}_{i+1}$ are identical unless the following two events occur:

  - The adversary does not query the random oracle on $\boldsymbol{\delta}$ in the first $i$ queries.
  - The adversary queries $\boldsymbol{\delta}$ on its $(i+1)^{\text{st}}$ query to the random oracle.

  However, if the adversary does not query $\boldsymbol{\delta}$ in its first $i$ queries, then by the above analysis, at the time of the adversary's $(i+1)^{\text{st}}$ query, the vector $\boldsymbol{\delta}$ is computationally hidden from its view. Thus the probability it queries $\boldsymbol{\delta}$ on its $(i+1)^{\text{st}}$ query is bounded by $1/2^{\lambda} + \mathsf{negl}(\lambda)$. By a hybrid argument, we conclude that an efficient adversary queries the random oracle on $\boldsymbol{\delta}$ with only negligible probability in $\mathsf{Hyb}_Q$, and in this case security holds.

**Our work: why broadcast complicates refresh.** While the above strategy is sound for plain public-key encryption, it does not directly extend to the distributed broadcast setting. In broadcast encryption, public keys are compressed into a single aggregated commitment, and the key update must be communicated *succinctly* to all of the users. A natural idea would be to reuse a common shift $\boldsymbol{\Delta}$ across all users in the group. This would allow for an efficient key update.

A standard entropy calculation shows that even if we leak $\{\mathbf{r}_i + \boldsymbol{\Delta}\}_{i \in S}$, there remains sufficient entropy in $\boldsymbol{\Delta}$ when $|S|$ is polynomial and the domain of $\mathbf{r}_i$ is large. However, as we show in Appendix A, sharing the same shift across multiple users enables a *zeroizing attack* on the resulting scheme. Since users can compute differences

$$(\mathbf{r}_1 + \boldsymbol{\Delta}) - (\mathbf{r}_2 + \boldsymbol{\Delta}) = \mathbf{r}_1 - \mathbf{r}_2,$$

an adversary can learn linear relations among the error terms *over the integers* and potentially break LWE security.

**Per-user randomized shifts.** To prevent this attack, our strategy is to derive independent *per-user* shifts using the random oracle. Namely, we now define the shift for user $i$ to be $\boldsymbol{\Delta}_i = H(\boldsymbol{\delta}, i)$. This ensures that shifts are independent across different users and avoids the zeroizing structure mentioned above.

The remaining challenge is to prove security in the broadcast setting while maintaining succinct aggregation. The proof combines two layers of hybrid arguments. We build on the selective proof of [WW25, §4]. There, the adversary first commits to a challenge recipient set $S^*$. The challenger then programs the public matrix $\mathbf{A}$ so that the contribution of users in $S^*$ is embedded into the LWE challenge instance. Ultimately, the reduction simulates the challenge ciphertext using the challenge from the decomposed LWE assumption. In the updatable setting, we must additionally integrate the refresh algorithm and forward security analysis into this framework. Our proof follows a similar structure to the basic version we sketched above for the case of vanilla dual Regev encryption, where we first rely on a smudging argument that leverages the min-entropy of the secret keys, then the leftover hash lemma to argue that each $\mathbf{B}\boldsymbol{\Delta}_i$ is statistically close to uniform, and finally, the decomposed LWE assumption to argue that the refresh randomness $\boldsymbol{\delta}$ is computationally hidden from the view of the adversary. We refer to Section 5 for the formal analysis.

**Implications to CGKA.** In Section 6, we finalize our main contributions by showing that:

- Incremental DBE (IncDBE) implies CGKA with post-compromise security.

- Updatable DBE (UpdDBE) implies CGKA with post-compromise security *and* forward secrecy.

While these implications are fairly immediate given the properties of these DBE schemes described above, for completeness, we provide an overview of the syntax, security, and efficiency properties of the CGKA schemes that we construct.

Our formalization of CGKA mostly follows conventions from [ACDT20] and [BDG+22]. Users are identified with a non-cryptographic ID (e.g., their phone number or email) as well as cryptographic public parameters pp and secret state st. Group evolution occurs as the result of Add, Remove, PCRefresh, and FSRefresh operations:

- Any user in the group with secret state st may run $\mathsf{Add}(\mathsf{st}, \mathsf{ID}, \mathsf{pp})$ using an external user's information $(\mathsf{ID}, \mathsf{pp})$ in order to update their own state, update the shared key $K$ (for the group), and produce a message $m$ that allows the user with ID to enter the group. Concretely, this message $m$ is processed by running a Join algorithm, which requires the public information $\{(\mathsf{ID}_j, \mathsf{pp}_j)\}_j$ associated with everyone currently in the group, and outputs a secret state together with the shared key $K$. Other users that are already in the group process $m$ using an algorithm Process, which also updates their state and key.
- Similarly, any user in the group with secret state st may run $\mathsf{Remove}(\mathsf{st}, \mathsf{ID}, \mathsf{pp})$ using an internal user's information $(\mathsf{ID}, \mathsf{pp})$ in order to remove ID from the group. This also results in a message $m$ that must be processed by all remaining members of the group in order to update their secret state and shared key.
- Finally, any user in the group may run $\mathsf{PCRefresh}(\mathsf{st})$ or $\mathsf{FSRefresh}(\mathsf{st})$, which generates a post-compromise or forward secrecy update, and produces a message $m$ that must be processed by all users in the group.

The constructions of CGKA from IncDBE or UpdDBE are natural. To Add or Remove a user, the members of the group perform the appropriate IncPK and IncSK algorithms, and then the invoking user samples a fresh $K$ and encrypts this key to the new aggregated public key. To implement PCRefresh, a user simply *removes* itself from the group, samples new public parameters, and then *adds* itself back to the group using these fresh parameters. To implement FSRefresh (in the case that we are starting with UpdDBE), the members of the group perform the key update algorithms of UpdDBE.

We model security against adversaries that are fully *selective* and *passive* (though we do allow the adversary to request that users stop deleting old states, following the standard conventions in, e.g., [ACDT20, BDG+22]). By passive, we mean that the adversary does not have the ability to craft maliciously generated messages on behalf of the participants or take any control over the network that is responsible for delivering messages. We further discuss this modeling choice and comparison with prior work at the end of Section 6.1. We define security using the "admissible query" paradigm, where an adversary is given access to oracles that control the evolution of group dynamics, including "corruption" and "challenge" oracles that allow the adversary to leak the state of users and ask for real-or-random challenges on the current group key. For any *fixed* sequence of oracle queries (note that this is a selective notion of security) that would not trivially allow the adversary to win, we demand that it has negligible advantage in its real-or-random queries. In particular, we consider the following two interesting types of query sequences:

1. The adversary first corrupts a user, then the user is either removed or refreshed, and then the adversary issues a challenge. This captures post-compromise security.
2. The adversary issues a challenge followed by a forward secrecy refresh, and *later* corrupts a user who was in the group at the time of the challenge. This captures forward secrecy.

In Section 6.1, we write down a set of admissible queries that captures only (1), and another that captures both (1) and (2). Then, in Section 6.2 and Section 6.3, we show that any adversary with noticeable advantage in the post-compromise CGKA game yields an adversary that can break the security of IncDBE (Definition 4.1), and any adversary with noticeable advantage in the post-compromise + forward secrecy CGKA game yields an adversary that can break the security of UpdDBE (Definition 5.1).

As mentioned above, our construction from IncDBE satisfies *optimal efficiency*, which we define as follows. There must be some fixed polynomial $p(\lambda)$ that bounds the run-time of each of Setup, Init, Add, Remove, PCRefresh, and Process, independent of the current size of the group. That is, the only operation that may depend on the size of the group is Join. This is in some sense inherent, if we consider reading the description of the group to be a part of the algorithm that users run to join the group.

Combining these details, we obtain our main theorem statement:

**Theorem 2.1** (Informal)**.** Assuming decomposed LWE [AMR25], there exist:

- A CGKA scheme with optimal efficiency that satisfies post-compromise security (against selective and passive adversaries).
- A CGKA scheme in the random oracle model that satisfies post-compromise security *and* forward secrecy (against selective and passive adversaries). Here, all computation and communication are optimal, except that the forward secrecy updates require a linear amount of *computation* in the group size.

# 3 Preliminaries

We write $\lambda \in \mathbb{N}$ to denote the security parameter. For a positive integer $n \in \mathbb{N}$, we write $[n] \coloneqq \{1, \ldots, n\}$. We write $\mathsf{poly}(\lambda)$ to denote a function that is $O(\lambda^c)$ for some constant $c \in \mathbb{N}$ and $\mathsf{negl}(\lambda)$ to denote a function that is negligible in $\lambda$ (i.e., $o(\lambda^{-c})$ for all $c \in \mathbb{N}$). For a finite set $S$, we write $x \xleftarrow{R} S$ to denote a uniform random draw from $S$. For a distribution $\mathcal{D}$, we write $x \leftarrow \mathcal{D}$ to denote a sample from $\mathcal{D}$. We say an algorithm is *efficient* if it runs in probabilistic polynomial time in the length of its input. Unless otherwise specified, all algorithms in this work should be assumed to be efficient. We say that two distribution ensembles $\mathcal{D}_0 = \{\mathcal{D}_{0,\lambda}\}_{\lambda\in\mathbb{N}}$ and $\mathcal{D}_1 = \{\mathcal{D}_{1,\lambda}\}_{\lambda\in\mathbb{N}}$ are *computationally indistinguishable* if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\left|\Pr[\mathcal{A}(1^\lambda, x) = 1 : x \leftarrow \mathcal{D}_{0,\lambda}] - \Pr[\mathcal{A}(1^\lambda, x) = 1 : x \leftarrow \mathcal{D}_{1,\lambda}]\right| = \mathsf{negl}(\lambda).$$

We say that $\mathcal{D}_0$ and $\mathcal{D}_1$ are *statistically indistinguishable* if the statistical distance between $\mathcal{D}_{0,\lambda}$ and $\mathcal{D}_{1,\lambda}$ is bounded by a negligible function $\mathsf{negl}(\lambda)$, and that they are *identical* if the statistical distance is 0. We write $\mathcal{D}_0 \approx_c \mathcal{D}_1$ to denote that $\mathcal{D}_0$ and $\mathcal{D}_1$ are computationally indistinguishable, and $\mathcal{D}_0 \approx_s \mathcal{D}_1$ to denote that they are statistically indistinguishable.

**Min-entropy.** Let $\mathcal{X}$ be a finite set and let $X$ be a random variable over $\mathcal{X}$ with probability mass function $p_X \colon \mathcal{X} \to [0, 1]$. The *min-entropy* of $X$ is defined as $H_\infty(X) = -\log(\max_{x\in\mathcal{X}} p_X(x))$. Let $(X, Y)$ be a pair of random variables over finite sets $\mathcal{X}$ and $\mathcal{Y}$ with joint distribution $p_{X,Y}(x, y) = \Pr[X = x, Y = y]$. The *average-case conditional min-entropy* of $X$ given $Y$ is defined to be

$$\widetilde{H}_\infty(X \mid Y) \coloneqq -\log\left(\mathbb{E}_{y\leftarrow Y}\left[\max_{x\in\mathcal{X}} \Pr[X = x \mid Y = y]\right]\right).$$

We denote $p_{\mathrm{guess}}(X \mid Y) \coloneqq \mathbb{E}_{y\leftarrow Y}[\max_{x\in\mathcal{X}} \Pr[X = x \mid Y = y]]$. Note that $\widetilde{H}_\infty(X \mid Y) = -\log p_{\mathrm{guess}}(X \mid Y)$.

## 3.1 Lattice Preliminaries

We follow the notation from [WW25]. We use bold uppercase letters (e.g., $\mathbf{A}, \mathbf{B}$) for matrices and bold lowercase letters (e.g., $\mathbf{u}, \mathbf{v}$) for vectors. By default, we represent vectors as column vectors (i.e., matrices with a single column). We use non-boldface letters for their components (e.g., $\mathbf{v} = [v_1, \ldots, v_n]^\intercal$). For a matrix $\mathbf{A} \in \mathbb{Z}^{n\times m}$, we write $\|\mathbf{A}\|$ to denote the maximum absolute value of its entries. When $\mathbf{A} \in \mathbb{Z}_q^{n\times m}$, we write $\|\mathbf{A}\|$ to denote $\|\tilde{\mathbf{A}}\|$ where $\tilde{\mathbf{A}}$ is the matrix obtained by lifting each entry of $\mathbf{A}$ to the integers (with values in the interval $(-q/2, q/2]$). We write $\mathbf{A} \otimes \mathbf{B}$ to denote the Kronecker product between matrices $\mathbf{A}$ and $\mathbf{B}$. For a matrix $\mathbf{A}$, we write $\mathsf{vec}(\mathbf{A})$ to denote the vector obtained by concatenating the columns of $\mathbf{A}$. For $x \in \mathbb{Z}_q$, we write $\lfloor x \rceil_2$ for the standard rounding function from $\mathbb{Z}_q$ to $\mathbb{Z}_2$, defined by

$$\lfloor x \rceil_2 \coloneqq \begin{cases} 0 & -q/4 \le x < q/4 \\ 1 & \text{otherwise.} \end{cases}$$

We extend $\lfloor \cdot \rceil_2$ to vectors and matrices via coordinate-wise evaluation.

**Leftover hash lemma.** We recall the following special case of the leftover hash lemma from [DRS04, ABB10].

**Lemma 3.1** (Leftover Hash Lemma [DRS04, ABB10])**.** Let $n, m, q \in \mathbb{N}$ where $m \geq 2(n+1)\log q$ and $q > 2$ is a prime. Then, for all polynomials $k = \mathsf{poly}(n)$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all fixed vectors $\mathbf{e} \in \mathbb{Z}_q^m$, the statistical distance between the following distributions is $\mathsf{negl}(n)$:

$$\left\{\begin{matrix}(\mathbf{A}, \mathbf{AK}, \mathbf{e}^\intercal\mathbf{K}) \\ \mathbf{A} \xleftarrow{R} \mathbb{Z}_q^{n\times m}, \mathbf{K} \xleftarrow{R} \{0,1\}^{m\times k}\end{matrix}\right\} \quad \text{and} \quad \left\{\begin{matrix}(\mathbf{A}, \mathbf{U}, \mathbf{e}^\intercal\mathbf{K}) \\ \mathbf{A} \xleftarrow{R} \mathbb{Z}_q^{n\times m}, \mathbf{U} \xleftarrow{R} \mathbb{Z}_q^{n\times k}, \mathbf{K} \xleftarrow{R} \{0,1\}^{m\times k}\end{matrix}\right\}.$$

**Discrete Gaussians.** We write $\mathcal{D}_{\mathbb{Z},\sigma}$ to denote the discrete Gaussian distribution over $\mathbb{Z}$ with width parameter $\sigma > 0$. We recall the following standard tail bound on the discrete Gaussian distribution.

**Lemma 3.2** (Gaussian Tail Bound)**.** For all $\lambda \in \mathbb{N}$ and $\sigma > 0$,

$$\Pr[|x| \geq \sqrt{\lambda}\sigma : x \leftarrow \mathcal{D}_{\mathbb{Z},\sigma}] \leq 2^{-\lambda}.$$

**Learning with errors and decomposed LWE.** The learning with errors (LWE) assumption [Reg05] with parameters $n, m, q, \widetilde{\sigma}$ asserts that the following two distributions are computationally indistinguishable:

$$(\mathbf{B}, \mathbf{s}^\intercal\mathbf{B} + \mathbf{e}^\intercal) \quad \text{and} \quad (\mathbf{B}, \mathbf{u}^\intercal), \tag{3.1}$$

where $\mathbf{B} \xleftarrow{R} \mathbb{Z}_q^{n\times m}$, $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$, $\mathbf{e} \leftarrow \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^m$, and $\mathbf{u} \xleftarrow{R} \mathbb{Z}_q^m$. In this work, we will use the matrix commitment scheme introduced in [Wee25]. Security of our scheme can in turn be based on either the succinct LWE assumption [Wee24] or the weaker decomposed LWE assumption [AMR25]. In this work, we will work exclusively with the decomposed LWE assumption since it enables constructions with *transparent* setup. Formally, the decomposed LWE assumption [AMR25] with parameters $n, m, q, \ell, \widetilde{\sigma}, \sigma$ asserts that Eq. (3.1) holds even when the matrix $\mathbf{B}$ is sampled in the following structured manner:

$$\mathbf{B} := \mathbf{W}(\mathbf{I}_\ell \otimes \mathbf{R}) + \mathsf{vec}(\mathbf{I}_\ell)^\intercal \otimes \mathbf{G} \quad \text{where} \quad \mathbf{W} \xleftarrow{R} \mathbb{Z}_q^{n\times \ell m},\ \mathbf{R} \leftarrow \mathcal{D}_{\mathbb{Z},\sigma}^{m\times \ell m}.$$

Here, $\mathsf{vec}(\mathbf{I}_\ell)$ denotes the vectorization of the identity matrix $\mathbf{I}_\ell$ (i.e., the vector obtained by concatenating the columns of $\mathbf{I}_\ell$). We now give the formal statement:

**Assumption 3.3** (Decomposed LWE)**.** Let $\lambda$ be a security parameter and let $n = n(\lambda), m = m(\lambda), q = q(\lambda), \widetilde{\sigma} = \widetilde{\sigma}(\lambda)$ be lattice parameters. Let $\sigma > 0$ be a Gaussian width parameter and $\ell \in \mathbb{N}$ be a dimension. We say that the $(\ell, \sigma)$-decomposed LWE assumption with parameters $(n, m, q, \widetilde{\sigma})$ holds if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\left|\Pr[\mathcal{A}(1^\lambda, \mathbf{W}, \mathbf{R}, \mathbf{s}^\intercal\mathbf{B} + \mathbf{e}^\intercal) = 1] - \Pr[\mathcal{A}(1^\lambda, \mathbf{W}, \mathbf{R}, \mathbf{u}^\intercal) = 1]\right| = \mathsf{negl}(\lambda),$$

where $\mathbf{B} = \mathbf{W}(\mathbf{I}_\ell \otimes \mathbf{R}) + \mathsf{vec}(\mathbf{I}_\ell)^\intercal \otimes \mathbf{G} \in \mathbb{Z}_q^{n\times \ell^2 m}$, $\mathbf{W} \xleftarrow{R} \mathbb{Z}_q^{n\times \ell m}$, $\mathbf{R} \leftarrow \mathcal{D}_{\mathbb{Z},\sigma}^{m\times \ell m}$, $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$, $\mathbf{e} \leftarrow \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\ell^2 m}$, and $\mathbf{u} \xleftarrow{R} \mathbb{Z}_q^{\ell^2 m}$. We will sometimes write "$\ell$-decomposed LWE" as shorthand to refer to an instance of the $(\ell, \sigma)$-decomposed LWE assumption for some (implicitly defined) width parameter $\sigma$.

The work of [CW26] shows that the leftover hash lemma also holds for the matrix $\mathbf{B}$ in the decomposed LWE assumption. Note that this does not follow from the standard leftover hash lemma (Lemma 3.1) because the matrix $\mathbf{B}$ in Assumption 3.3 is not sampled uniformly at random. However, $\mathbf{B}$ contains sub-matrices that are statistically close to uniform, which is sufficient for the leftover hash lemma. We give the specific statement from [CW26, Corollary 2.9] here:

**Lemma 3.4** (Leftover Hash Lemma for Decomposed LWE Matrix $\mathbf{B}$ [CW26, Corollary 2.9])**.** Let $n, m, q$ be integers such that $m \geq 2(n+1)\log q$ and $q$ is prime. Let $\ell \in \mathbb{N}$, $\hat{m} = \ell^2 m$, and suppose $\sigma \geq \log m$. Then, for all fixed vectors $\mathbf{e} \in \mathbb{Z}_q^{\hat{m}}$ and all $k = \mathsf{poly}(n)$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that the statistical distance between the following distributions is $\mathsf{negl}(n)$:

- $\left\{(\mathbf{W}, \mathbf{R}, \mathbf{BK},\ \mathbf{e}^\intercal\mathbf{K}) \ :\ \mathbf{W} \xleftarrow{R} \mathbb{Z}_q^{n\times \ell m},\ \mathbf{R} \leftarrow \mathcal{D}_{\mathbb{Z},\sigma}^{m\times \ell m},\ \mathbf{K} \xleftarrow{R} \{0,1\}^{\hat{m}\times k}\right\}$; and
- $\left\{(\mathbf{W}, \mathbf{R}, \mathbf{U},\ \mathbf{e}^\intercal\mathbf{K}) \ :\ \mathbf{W} \xleftarrow{R} \mathbb{Z}_q^{n\times \ell m},\ \mathbf{R} \leftarrow \mathcal{D}_{\mathbb{Z},\sigma}^{m\times \ell m},\ \mathbf{U} \xleftarrow{R} \mathbb{Z}_q^{n\times k},\ \mathbf{K} \xleftarrow{R} \{0,1\}^{\hat{m}\times k}\right\}$,

where $\mathbf{B} = \mathbf{W}(\mathbf{I}_\ell \otimes \mathbf{R}) + \mathsf{vec}(\mathbf{I}_\ell)^\intercal \otimes \mathbf{G} \in \mathbb{Z}_q^{n\times \hat{m}}$ in both distributions.

**Matrix commitments.** Similar to [WW25], the core building block of our constructions is the matrix commitments introduced by Wee [Wee25]. Specifically, we use the one for committing to sparse matrices from [WW25]. Here, there are algorithms that take as input a wide matrix $\mathbf{M} \in \mathbb{Z}_q^{n \times L}$ and output a commitment $\mathbf{C} \in \mathbb{Z}_q^{n \times m}$ and a low-norm opening $\mathbf{Z} \in \mathbb{Z}_q^{4m^5 \times L}$ such that

$$\mathbf{C}\mathbf{V}_L = \mathbf{M} - \mathbf{B}\mathbf{Z}.$$

The matrices $\mathbf{B}$ and $\mathbf{V}_L$ are fixed matrices determined by a set of public parameters pp, and moreover, the matrix $\mathbf{V}_L$ is low-norm. As in [WW25], we will consider committing to *sparse* matrices with an exponential total number of columns $L = 2^\lambda$, but only a polynomial number of *non-zero* columns. The work of [WW25] shows that there are efficient algorithms to both compute the commitment to the matrix $\mathbf{M}$ as well as any column of $\mathbf{V}_L$ and $\mathbf{Z}$ in time that scales polynomially with the number of non-zero columns of $\mathbf{M}$ (rather than the total number of columns in $\mathbf{M}$). We now give the formal syntax and efficiency properties (when instantiated with the decomposed LWE assumption):

**Fact 3.5** (Sparse Matrix Commitment [Wee25, WW25, adapted])**.** Let $\lambda$ be a security parameter and $n, m, q, \widetilde{\sigma}, \sigma$ be lattice parameters (which can be functions of the security parameter $\lambda$). There exists a tuple of efficient algorithms $(\mathsf{Setup}, \mathsf{Com}, \mathsf{Ver}, \mathsf{Open})$ with the following syntax and properties:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}_{\mathsf{com}}$: On input the security parameter $\lambda$, the setup algorithm outputs a set of public parameters $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$ where $\mathbf{W} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n \times 2m^3}$, $\mathbf{R} \in \mathbb{Z}^{m \times 2m^3}$, and $\mathbf{B} = \mathbf{W}(\mathbf{I}_{2m^2} \otimes \mathbf{R}) + \mathrm{vec}(\mathbf{I}_{2m^2})^\intercal \otimes \mathbf{G} \in \mathbb{Z}_q^{n \times 4m^5}$. We note that $\|\mathbf{R}\| \le \sqrt{\lambda} \cdot \sigma$.[9]
- $\mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{M}) \to \mathbf{C}$: On input the public parameters $\mathsf{pp}_{\mathsf{com}}$ and a (sparse) matrix $\mathbf{M} \in \mathbb{Z}_q^{n \times L}$ where $L \le 2^\lambda$, the Com algorithm outputs a commitment $\mathbf{C} \in \mathbb{Z}_q^{n \times m}$. This algorithm is deterministic.
- $\mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i) \to \mathbf{v}_i$: On input the public parameters $\mathsf{pp}_{\mathsf{com}}$ and an index $i \in [2^\lambda]$, the Ver algorithm outputs a vector $\mathbf{v}_i \in \mathbb{Z}_q^m$. This algorithm is deterministic.
- $\mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{M}, i) \to \mathbf{z}_i$: On input the public parameters $\mathsf{pp}_{\mathsf{com}}$, a matrix $\mathbf{M} \in \mathbb{Z}_q^{n \times L}$ where $L \le 2^\lambda$, and an index $i \in [2^\lambda]$, the Open algorithm outputs a vector $\mathbf{z}_i \in \mathbb{Z}_q^{4m^5}$. This algorithm is deterministic.

The matrix commitment scheme satisfies the following properties:

- **Efficiency:** Algorithms $\mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{M})$ and $\mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{M}, \cdot)$ run in time $\mathrm{poly}(\lambda, K, m, \log q)$ where $K$ is the number of non-zero columns of $\mathbf{M}$. Algorithm $\mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, \cdot)$ runs in time $\mathrm{poly}(\lambda, m, \log q)$.
- **Correctness:** For all security parameters $\lambda \in \mathbb{N}$, all public parameters $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$ in the support of $\mathsf{Setup}(1^\lambda)$, all matrices $\mathbf{M} = [\mathbf{m}_1 \mid \cdots \mid \mathbf{m}_L] \in \mathbb{Z}_q^{n \times L}$ where $L = 2^\lambda$, and all indices $i \in [2^\lambda]$, if we set

$$\begin{aligned} \mathbf{C} &= \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{M}) \\ \mathbf{v}_i &= \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i) \\ \mathbf{z}_i &= \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{M}, i), \end{aligned}$$

  the following holds:

$$\mathbf{C}\mathbf{v}_i = \mathbf{m}_i - \mathbf{B}\mathbf{z}_i. \tag{3.2}$$

  Moreover, $\|\mathbf{v}_i\|$ and $\|\mathbf{z}_i\|$ satisfy the following norm bounds:

$$\|\mathbf{v}_i\| \le O(\|\mathbf{R}\| \cdot m^4 \log q) \quad \text{and} \quad \|\mathbf{z}_i\| \le O(\|\mathbf{R}\| \cdot m^7 \log q \log L).$$

  Since $\mathsf{Setup}(1^\lambda)$ always outputs $\mathbf{R} \in \mathbb{Z}^{m \times 2m^3}$ where $\|\mathbf{R}\| \le \sigma\sqrt{\lambda}$, this means that for all $L \le 2^\lambda$, we can bound

$$\begin{aligned} \|\mathbf{v}_i\| &\le O(\|\mathbf{R}\| \cdot m^4 \log q) \le O(\sigma\sqrt{\lambda} \cdot m^4 \log q) = O(m^4 \sigma \log q \cdot \sqrt{\lambda}), \\ \|\mathbf{z}_i\| &\le O(\|\mathbf{R}\| \cdot m^7 \log q \log L) \le O(\sigma\sqrt{\lambda} \cdot m^7 \log q \cdot \lambda) = O(m^7 \sigma \log q \cdot \lambda^{3/2}). \end{aligned}$$

[9] Strictly speaking, $\mathbf{R} \leftarrow \mathcal{D}_{\mathbb{Z},\sigma}^{m \times 2m^3}$, so this property only holds with overwhelming probability. However, we can sample $\mathbf{R}$ from a truncated discrete Gaussian distribution so the magnitude of the entries is bounded by $\sqrt{\lambda} \cdot \sigma$. This does not affect correctness, and only introduces a negligible loss to security.

- **Security:** Under the $(2m^2, \sigma)$-decomposed LWE assumption with parameters $(n, m, q, \widetilde{\sigma})$, for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\left|\Pr[\mathcal{A}(1^\lambda, \mathsf{pp}_{\mathsf{com}}, \mathbf{s}^\intercal\mathbf{B} + \mathbf{e}^\intercal) = 1] - \Pr[\mathcal{A}(1^\lambda, \mathsf{pp}_{\mathsf{com}}, \mathbf{u}^\intercal) = 1]\right| = \mathsf{negl}(\lambda),$$

  where $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B}) \leftarrow \mathsf{Setup}(1^\lambda)$, $\mathbf{s} \xleftarrow{R} \mathbb{Z}_q^n$, $\mathbf{e} \leftarrow \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\hat{m}}$, and $\mathbf{u} \xleftarrow{R} \mathbb{Z}_q^{\hat{m}}$, where $\hat{m} = (2m^2)^2 \cdot m = 4m^5$.

# 4 Incremental Distributed Broadcast Encryption

In this section, we introduce our notion of incremental distributed broadcast encryption, which serves as the cryptographic core of our group-messaging scheme. We start by introducing the syntax and functionality requirements of the primitive. At a high level, an incremental distributed broadcast encryption scheme consists of a tuple of efficient algorithms $\Pi_{\mathsf{IncDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec})$. In Section 6, we show formally how to use an incremental distributed broadcast encryption scheme to construct a continuous group key agreement scheme. Here, we start with a high-level description of how these algorithms map to a group messaging setting.

- **Setup and key generation.** A system administrator runs Setup to generate a set of public parameters pp. We assume each user in the system is associated with a distinct index $i$. Each user $i$ can generate their individual key-pair $(\mathsf{pk}_i, \mathsf{sk}_i)$ by running $\mathsf{KeyGen}(\mathsf{pp}, i)$.
- **Updating keys when users join or leave the system.** Whenever a new user joins the system or an existing user leaves the system, each user $i$ in the system uses the IncPK and IncSK algorithms to update the global (public) encryption key apk as well as their individual secret key $\mathsf{ask}_i$. All of the users share the global encryption key apk (which is an aggregate public key derived from the public keys of all of the users in the group). Each user's individual secret key $\mathsf{ask}_i$ is a function of both the user's secret key $\mathsf{sk}_i$ as well as the public keys of the other users in the group. The "incremental" property of the scheme requires that the amount of work each user has to perform to update the public key apk and their local decryption key $\mathsf{ask}_i$ depends only on the number of users that are joining or leaving the system, and *not* the total number of users in the system.
- **Encryption and decryption.** To send a message $\mu$ to the current set of users, a sender (in the group) runs the encryption algorithm Enc using the current public key apk. The encryption algorithm outputs a ciphertext ct. Upon receiving ct, each recipient $i$ runs the decryption algorithm Dec with their secret key $\mathsf{ask}_i$. The decryption algorithm outputs the plaintext message.

We now give the formal definition.

**Definition 4.1** (Incremental Distributed Broadcast Encryption)**.** An incremental distributed broadcast encryption scheme is a tuple of efficient algorithms $\Pi_{\mathsf{IncDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec})$ with the following syntax:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$: On input the security parameter $\lambda$, the setup algorithm outputs a set of public parameters pp.
- $\mathsf{KeyGen}(\mathsf{pp}, i) \to (\mathsf{pk}_i, \mathsf{sk}_i)$: On input the public parameters pp and an index $i \in [2^\lambda]$, the key-generation algorithm outputs a public key $\mathsf{pk}_i$ and a secret key $\mathsf{sk}_i$.
- $\mathsf{IncPK}(\mathsf{pp}, \mathsf{apk}, (i, \mathsf{pk}_i), \mathsf{op}) \to \mathsf{apk}'$: On input the public parameters pp, an aggregated public key apk, a public key $\mathsf{pk}_i$ (with index $i$), and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, the update-public-key algorithm computes a new aggregated public key $\mathsf{apk}'$. This algorithm is deterministic.
- $\mathsf{IncSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, (i, \mathsf{pk}_i), \mathsf{op}) \to \mathsf{ask}'_{i^*}$: On input the public parameters pp, an aggregated secret key $\mathsf{ask}_{i^*}$ for user $i^*$, a public key $\mathsf{pk}_i$ (with index $i$), and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, the update-secret-key algorithm computes an updated aggregated secret key $\mathsf{ask}'_{i^*}$. This algorithm is deterministic.
- $\mathsf{Enc}(\mathsf{pp}, \mathsf{apk}, \mu) \to \mathsf{ct}$: On input the public parameters pp, an aggregated public key apk, and a message $\mu \in \{0, 1\}$, the encryption algorithm outputs a ciphertext ct.
- $\mathsf{Dec}(\mathsf{pp}, \mathsf{ask}_i, \mathsf{ct}) \to \mu$: On input the public parameters pp, the aggregated secret key $\mathsf{ask}_i$, and a ciphertext ct, the decryption algorithm outputs a message $\mu \in \{0, 1\}$.

**Correctness.** The correctness requirement for an incremental DBE scheme states that for any sequence of add and remove operations, whenever a user encrypts a message to the current group, all users currently in the group can decrypt it using their respective aggregated secret keys. To formally define correctness, we define two helper subroutines: (1) an UpdateAPK subroutine that tracks the current aggregated public key associated with the group; and (2) an UpdateASK algorithm that tracks the aggregated secret key associated with a particular user. We use these subroutines to streamline our correctness and security definitions.

In more detail, the UpdateAPK subroutine takes the public parameters pp, the current aggregated public key associated with the group (initially $\perp$), the public key $\mathsf{pk}_i$ associated with a user who is either joining or leaving the group, and a mapping $S$ representing the current members of the group (i.e., $S$ maps an index $j$ to a public key $\mathsf{pk}_j$). The UpdateAPK subroutine then outputs the updated aggregated public key and the updated membership mapping after the operation. Throughout, we write $i \in S$ to denote that the index $i$ is in the domain of the mapping $S$, and we write $\varnothing$ to denote the empty mapping. We define the syntax formally below:

**Definition 4.2** (UpdateAPK Subroutine for Incremental DBE)**.** Let $\Pi_{\mathsf{IncDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec})$ be an incremental distributed broadcast encryption scheme. We define the UpdateAPK subroutine as follows:

$\mathsf{UpdateAPK}(\mathsf{pp}, \mathsf{apk}, (i, \mathsf{pk}_i), \mathsf{op}, S)$

**Inputs:** Public parameters pp, an aggregated public key apk, a public key $\mathsf{pk}_i$, an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, and a mapping $S$ that maps an index $j$ (of a group member) to a public key $\mathsf{pk}_j$.

**Outputs:** Updated aggregated public key apk and updated mapping $S$.

**Computation:**

- If $\mathsf{op} = \mathsf{Add}$ and $i \notin S$, compute $\mathsf{apk} = \mathsf{IncPK}(\mathsf{pp}, \mathsf{apk}, (i, \mathsf{pk}_i), \mathsf{Add})$. Set $S[i] = \mathsf{pk}_i$ and return $(\mathsf{apk}, S)$.
- If $\mathsf{op} = \mathsf{Remove}$ and $S[i] = \mathsf{pk}_i$, compute $\mathsf{apk} = \mathsf{IncPK}(\mathsf{pp}, \mathsf{apk}, (i, \mathsf{pk}_i), \mathsf{Remove})$. Remove $i$ from $S$ and return $(\mathsf{apk}, S)$.
- In all other cases, return $(\mathsf{apk}, S)$.

Next, we define the corresponding UpdateASK subroutine that keeps track of a user's aggregate secret key as users join or leave the group. The UpdateASK subroutine takes the following inputs:

- the public parameters pp,
- the current set of group members $S$, represented as a mapping from the index $j$ of a group member to their public key $\mathsf{pk}_j$,
- the secret key $\mathsf{sk}_{i^*}$ and the aggregated secret key $\mathsf{ask}_{i^*}$ for the user performing the update (i.e., user $i^*$),
- the public key $\mathsf{pk}_i$ of the user joining/leaving the group, and
- the description of the operation op (i.e., specifying whether user $i$ is joining or leaving the group).

The UpdateASK subroutine outputs the updated secret key $\mathsf{ask}_{i^*}$ associated with user $i^*$. In the correctness and security definitions, the challenger uses UpdateASK to track the aggregated secret key $\mathsf{ask}_{i^*}$ for each user. We now describe the formal syntax and behavior and then give our formal correctness definition:

**Definition 4.3** (UpdateASK Subroutine for Incremental DBE)**.** Let $\Pi_{\mathsf{IncDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec})$ be an incremental distributed broadcast encryption scheme. We define the UpdateASK subroutine as follows:

$\mathsf{UpdateASK}(\mathsf{pp}, (i^*, \mathsf{sk}_{i^*}, \mathsf{ask}_{i^*}), (i, \mathsf{pk}_i), \mathsf{op}, S)$

**Inputs:** Public parameters $\mathsf{pp}$, a secret key $\mathsf{sk}_{i^*}$ and aggregated secret key $\mathsf{ask}_{i^*}$ for user $i^*$, a public key $\mathsf{pk}_i$ for user $i$, an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, and a mapping $S$ that maps an index $j$ (of a group member) to a public key $\mathsf{pk}_j$.

**Outputs:** Updated aggregated secret key $\mathsf{ask}_{i^*}$.

**Computation:**

- Suppose $\mathsf{op} = \mathsf{Add}$ and $i \notin S$. We consider two cases:
  - If $i^* \in S$, return the updated key $\mathsf{ask}_{i^*} = \mathsf{IncSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, (i, \mathsf{pk}_i), \mathsf{Add})$.
  - If $i^* = i$, then initialize $\mathsf{ask}_{i^*} = \mathsf{sk}_{i^*}$. Then, for each index $j \in S$ in lexicographic order, run $\mathsf{ask}_{i^*} = \mathsf{IncSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, (j, S[j]), \mathsf{Add})$. Return $\mathsf{ask}_{i^*}$.
- Suppose $\mathsf{op} = \mathsf{Remove}$ and $S[i] = \mathsf{pk}_i$. We consider two cases:
  - If $i^* \in S$ and $i^* \neq i$, return the updated key $\mathsf{ask}_{i^*} = \mathsf{IncSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, (i, \mathsf{pk}_i), \mathsf{Remove})$.
  - If $i^* = i$, return $\mathsf{ask}_{i^*} = \perp$.
- In all other cases (in particular, whenever $i^* \notin S \cup \{i\}$), return $\mathsf{ask}_{i^*}$ unchanged.

**Definition 4.4** (Correctness)**.** Let $\Pi_{\mathsf{IncDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec})$ be an incremental distributed broadcast encryption scheme. Let $\lambda \in \mathbb{N}$ be a security parameter, $Q \in \mathbb{N}$ be the number of queries, $\mathsf{pp}$ be a set of public parameters, and $\{(\mathsf{ind}_j, \mathsf{pk}_j, \mathsf{op}_j)\}_{j \in [Q]}$ be a sequence of queries where $\mathsf{op}_j \in \{\mathsf{Add}, \mathsf{Remove}\}$. For an index $i^* \in [2^\lambda]$ and a public/secret key-pair $(\mathsf{pk}_{i^*}, \mathsf{sk}_{i^*})$, we define the triple $(\mathsf{apk}_Q, \mathsf{ask}_{i^*,Q}, S_Q)$ associated with this sequence as follows:

- Initialize $\mathsf{apk}_0 = \perp$, $\mathsf{ask}_{i^*,0} = \perp$, and $S_0 = \varnothing$.
- For each $j \in [Q]$, compute

$$(\mathsf{apk}_j, S_j) = \mathsf{UpdateAPK}(\mathsf{pp}, \mathsf{apk}_{j-1}, (\mathsf{ind}_j, \mathsf{pk}_j), \mathsf{op}_j, S_{j-1})$$
$$\mathsf{ask}_{i^*,j} = \mathsf{UpdateASK}(\mathsf{pp}, (i^*, \mathsf{sk}_{i^*}, \mathsf{ask}_{i^*,j-1}), (\mathsf{ind}_j, \mathsf{pk}_j), \mathsf{op}_j, S_{j-1}).$$

- Output $(\mathsf{apk}_Q, \mathsf{ask}_{i^*,Q}, S_Q)$.

We say $\Pi_{\mathsf{IncDBE}}$ is correct if for all security parameters $\lambda \in \mathbb{N}$, all $\mathsf{pp}$ in the support of $\mathsf{Setup}(1^\lambda)$, all $Q \leq 2^\lambda$, all indices $i^* \in [2^\lambda]$, all $(\mathsf{pk}_{i^*}, \mathsf{sk}_{i^*})$ in the support of $\mathsf{KeyGen}(\mathsf{pp}, i^*)$, all sequences $\{(\mathsf{ind}_j, \mathsf{pk}_j, \mathsf{op}_j)\}_{j \in [Q]}$ where $S_Q[i^*] = \mathsf{pk}_{i^*}$, and all messages $\mu \in \{0, 1\}$, we have

$$\Pr[\mathsf{Dec}(\mathsf{pp}, \mathsf{ask}_{i^*,Q}, \mathsf{ct}) = \mu : \mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{apk}_Q, \mu)] = 1,$$

where $(\mathsf{apk}_Q, \mathsf{ask}_{i^*,Q}, S_Q)$ is the triple associated with the sequence $\{(\mathsf{ind}_j, \mathsf{pk}_j, \mathsf{op}_j)\}_{j \in [Q]}$.

**Succinctness.** Our succinctness requirement stipulates that the size of the aggregate public key apk and the size of the aggregate secret key ask output by IncPK and IncSK are short (i.e., have size that is independent of the size of the group).

**Definition 4.5** (Succinctness)**.** There exists a universal polynomial $\mathsf{poly}(\cdot)$ such that for all security parameters $\lambda \in \mathbb{N}$, all public parameters $\mathsf{pp}$ in the support of $\mathsf{Setup}(1^\lambda)$, the size of the aggregated public key apk output by $\mathsf{IncPK}(\mathsf{pp}, \cdot, \cdot, \cdot)$ and the size of the aggregated secret key ask output by $\mathsf{IncSK}(\mathsf{pp}, \cdot, \cdot, \cdot)$ are bounded by $\mathsf{poly}(\lambda)$.

**Incremental DBE security.** We now define semantic security for incremental distributed broadcast encryption. In the *adaptive* definition, an adversary may dynamically add or remove users from the group. In the *static* variant, we require the adversary to commit to its entire sequence of add/remove operations at the start of the game. In both cases, we require semantic security for any ciphertext encrypted with respect to the group public key provided that the adversary has not corrupted any user in the final group. We now give the formal definition.

**Definition 4.6** (Adaptive Security)**.** Let $\Pi_{\mathsf{IncDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec})$ be an incremental distributed broadcast encryption scheme. For a security parameter $\lambda \in \mathbb{N}$, an adversary $\mathcal{A}$, and a challenge bit $b \in \{0,1\}$, we define the incremental distributed broadcast encryption security game as follows:

- **Setup phase:** The challenger starts by sampling $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ and initializes a counter $\mathsf{ctr} = 0$ and an (initially empty) dictionary D to keep track of key-generation queries. It also initializes a mapping $S = \emptyset$ that maps the index of each member of the current broadcast set to their public key, a set $T = \emptyset$ to keep track of the counters associated with the current broadcast set, the initial aggregated public key $\mathsf{apk} = \perp$, and a set of corrupted counters $C = \emptyset$.
- **Pre-challenge query phase:** The adversary may issue the following queries:
  - **Key-generation queries:** On input an index $i \in [2^\lambda]$, the challenger increments the counter $\mathsf{ctr} = \mathsf{ctr} + 1$ and samples $(\mathsf{pk}_{\mathsf{ctr}}, \mathsf{sk}_{\mathsf{ctr}}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}, i)$. It also initializes an aggregate decryption key $\mathsf{ask}_{\mathsf{ctr}} = \perp$ and adds the mapping $\mathsf{ctr} \mapsto (i, \mathsf{pk}_{\mathsf{ctr}}, \mathsf{sk}_{\mathsf{ctr}}, \mathsf{ask}_{\mathsf{ctr}})$ to D. The challenger gives $\mathsf{pk}_{\mathsf{ctr}}$ to the adversary $\mathcal{A}$.
  - **Corruption queries:** On input a counter value $c \in [\mathsf{ctr}]$, the challenger looks up $(i, \mathsf{pk}_c, \mathsf{sk}_c, \mathsf{ask}_c) = \mathsf{D}[c]$ and returns $(\mathsf{sk}_c, \mathsf{ask}_c)$. The challenger then updates $C = C \cup \{c\}$.
  - **Update queries:** On input a counter value $c \in [\mathsf{ctr}]$ and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, the challenger looks up $(i, \mathsf{pk}_c, \mathsf{sk}_c, \mathsf{ask}_c) = \mathsf{D}[c]$. The challenger then checks the following conditions and responds with $\perp$ if either holds:
    - **Associating a key with an existing index** $i$**:** $\mathsf{op} = \mathsf{Add}$ and $i \in S$;
    - **Removing a public key that is not in the group:** $\mathsf{op} = \mathsf{Remove}$ and $c \notin T$.

    Otherwise, the challenger proceeds as follows:
    - The challenger first computes the updated aggregated public key and membership mapping
    $$(\mathsf{apk}', S') = \mathsf{UpdateAPK}(\mathsf{pp}, \mathsf{apk}, (i, \mathsf{pk}_c), \mathsf{op}, S).$$
    - For each $t \in T \cup \{c\}$, the challenger looks up $(i_t, \mathsf{pk}_t, \mathsf{sk}_t, \mathsf{ask}_t) = \mathsf{D}[t]$. Next, for each $t \in T \cup \{c\}$, the challenger computes the updated aggregate decryption key
    $$\mathsf{ask}_t = \mathsf{UpdateASK}\big(\mathsf{pp}, (i_t, \mathsf{sk}_t, \mathsf{ask}_t), (i, \mathsf{pk}_c), \mathsf{op}, S\big),$$
    using the previous mapping $S$ (i.e., the mapping from *before* the current operation). The challenger updates $\mathsf{D}[t] = (i_t, \mathsf{pk}_t, \mathsf{sk}_t, \mathsf{ask}_t)$.
    - The challenger then updates $\mathsf{apk} = \mathsf{apk}'$ and $S = S'$.
    - If $\mathsf{op} = \mathsf{Add}$, then the challenger updates $T = T \cup \{c\}$. If $\mathsf{op} = \mathsf{Remove}$, then it updates $T = T \setminus \{c\}$.

    The challenger responds with the updated aggregate public key apk as well as the updated aggregate decryption key $\mathsf{ask}_t$ for all corrupted users $t \in T \cap C$.
- **Challenge phase:** When the adversary is finished making queries and proceeds to the challenge phase, the challenger starts by checking if $T = \emptyset$ or if $T \cap C \neq \emptyset$. In either case, the challenger halts the experiment with output 0. Otherwise, the challenger responds with the challenge ciphertext $\mathsf{ct}^* \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{apk}, b)$. The challenger also sets $T^* = T$ to remember the set of counters associated with the challenge ciphertext.
- **Post-challenge query phase:** Same as the pre-challenge query phase above, except the adversary is no longer allowed to make a corruption query on any counter $c \in T^*$.
- **Output phase:** At the end of the experiment, the adversary outputs a bit $b' \in \{0,1\}$, which is the output of the experiment.

We say an incremental DBE scheme is *secure* if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,
$$\Big|\Pr[b' = 1 : b = 0] - \Pr[b' = 1 : b = 1]\Big| \leq \mathsf{negl}(\lambda),$$
where $b' \in \{0,1\}$ is the output of an execution of the above security experiment.

**Remark 4.7** (Registering Malicious Keys)**.** In Definition 4.6, the challenger generates the public keys for each user honestly and the adversary has the ability to pick the subset of keys that are added to the group. We could consider a stronger definition that would allow the adversary to choose the public keys for *corrupted* users. As long as none of these corrupted users are present in the group during the challenge phase, we could still expect security to hold. Giving the adversary the ability to choose its own public keys is a standard requirement in registration-based cryptographic schemes as well as other trustless cryptographic schemes (e.g., [GHMR18, HLWW23, GKPW24, CW26]). In the context of group messaging, this stronger notion would correspond to security against insider attacks [AJM22]. In this work (specifically, in Section 6), we focus on the weaker notion of security where we assume that group members generate their keys honestly, and security is required only against a passive adversary that may corrupt users' states but does not actively inject malicious keys into the group. For this reason, we opt here for the weaker security definition for incremental distributed broadcast encryption that does not allow the adversary to choose the public keys for corrupted users.

**Remark 4.8** (Multi-Bit Messages)**.** While we have defined the syntax and security of incremental DBE for encrypting bits, it is straightforward to support longer messages by simply encrypting bit by bit. In the security definition, we would now allow the adversary to submit two equal-length messages $\mu_0, \mu_1$ during the challenge phase and receive an encryption of $\mu_b$.

**Definition 4.9** (Static Security)**.** The *static* security game for an incremental DBE scheme $\Pi_{\mathsf{IncDBE}}$ is identical to the security game from Definition 4.6, except the adversary commits to all of its pre-challenge queries (i.e., key-generation queries, corruption queries, and update queries) at the beginning of the experiment. The post-challenge query phase is unchanged. We say $\Pi_{\mathsf{IncDBE}}$ satisfies *static* security if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\left|\Pr[b' = 1 : b = 0] - \Pr[b' = 1 : b = 1]\right| \leq \mathsf{negl}(\lambda),$$

where $b' \in \{0, 1\}$ is the output of an execution of the static security experiment.

## 4.1 Constructing Incremental Distributed Broadcast Encryption

Our construction relies on the matrix commitment scheme for sparse matrices from Fact 3.5. To obtain our incremental distributed broadcast encryption scheme, we rely on the fact that the scheme in Fact 3.5 is incremental. We show this formally below and then give our incremental distributed broadcast encryption scheme.

**Claim 4.10** (Linearity of Matrix Commitment)**.** Let $\lambda$ be a security parameter and $n, m, q, \widetilde{\sigma}, \sigma$ be lattice parameters. Let $(\mathsf{Setup}, \mathsf{Com}, \mathsf{Ver}, \mathsf{Open})$ be the sparse matrix commitment scheme from Fact 3.5. Take any $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$ in the support of $\mathsf{Setup}(1^\lambda)$, any collection of matrices $\mathbf{M}_1, \ldots, \mathbf{M}_k \in \mathbb{Z}_q^{n \times L}$ where $L \leq 2^\lambda$, any collection of scalars $\alpha_1, \ldots, \alpha_k \in \{-1, 0, 1\}$, and any index $i \in [2^\lambda]$. Let

$$\begin{aligned}
\mathbf{M} &= \sum_{j \in [k]} \alpha_j \cdot \mathbf{M}_j, \\
\mathbf{C} &= \sum_{j \in [k]} \alpha_j \cdot \mathbf{C}_j \quad \text{where} \quad \mathbf{C}_j = \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{M}_j), \\
\mathbf{z}_i &= \sum_{j \in [k]} \alpha_j \cdot \mathbf{z}_{j,i} \quad \text{where} \quad \mathbf{z}_{j,i} = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{M}_j, i), \\
\mathbf{v}_i &= \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i).
\end{aligned}$$

Then, $\mathbf{C}\mathbf{v}_i = \mathbf{m}_i - \mathbf{B}\mathbf{z}_i$, where $\mathbf{m}_i$ is the $i^{\text{th}}$ column of $\mathbf{M}$. Moreover, $\|\mathbf{v}_i\| \leq O(\|\mathbf{R}\| \cdot m^4 \log q)$ and $\|\mathbf{z}_i\| \leq O(k \cdot \|\mathbf{R}\| \cdot m^7 \log q \log L)$.

*Proof.* By the correctness of the matrix commitment scheme (Eq. (3.2)), for each $j \in [k]$ we have

$$\mathbf{C}_j \mathbf{v}_i = \mathbf{m}_{j,i} - \mathbf{B}\mathbf{z}_{j,i},$$

where $\mathbf{m}_{j,i}$ is the $i^{\text{th}}$ column of $\mathbf{M}_j$. Thus, we get

$$\mathbf{C}\mathbf{v}_i = \left(\sum_{j\in[k]} \alpha_j \mathbf{C}_j\right)\mathbf{v}_i = \sum_{j\in[k]} \left(\alpha_j(\mathbf{m}_{j,i} - \mathbf{B}\mathbf{z}_{j,i})\right) = \sum_{j\in[k]} \left(\alpha_j \mathbf{m}_{j,i}\right) - \mathbf{B}\sum_{j\in[k]} \left(\alpha_j \mathbf{z}_{j,i}\right) = \mathbf{m}_i - \mathbf{B}\mathbf{z}_i.$$

For the norm bounds, the bound on $\|\mathbf{v}_i\|$ is immediate from Fact 3.5. For $\|\mathbf{z}_i\|$, the vector $\mathbf{z}_i$ is a linear combination of at most $k$ vectors $\mathbf{z}_{j,i}$, each of norm $\|\mathbf{z}_{j,i}\| \le O(\|\mathbf{R}\| \cdot m^7 \log q \log L)$ by Fact 3.5. The claim then follows by the triangle inequality. □

**Construction 4.11** (Incremental Distributed Broadcast Encryption)**.** Let $\lambda$ be a security parameter and $n, m, q, \widetilde{\sigma}, \sigma$ be lattice parameters (as in Fact 3.5). Let $\hat{m} = 4m^5$ and $\xi = \lfloor q/2 \rfloor$. Let $(\mathsf{Setup}_{\mathsf{com}}, \mathsf{Com}, \mathsf{Ver}, \mathsf{Open})$ denote the algorithms of the sparse matrix commitment scheme from Fact 3.5. We construct an incremental distributed broadcast encryption scheme $\Pi_{\mathsf{IncDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec})$ as follows:

- $\mathsf{Setup}(1^\lambda)$: On input the security parameter $\lambda$, the setup algorithm samples the matrix commitment parameters $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B}) \leftarrow \mathsf{Setup}_{\mathsf{com}}(1^\lambda)$, together with $\mathbf{A} \xleftarrow{\text{R}} \mathbb{Z}_q^{n\times m}$ and $\mathbf{p} \xleftarrow{\text{R}} \mathbb{Z}_q^n$. It outputs $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$.
- $\mathsf{KeyGen}(\mathsf{pp}, i)$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$ where $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$ and an index $i \in [2^\lambda]$, the key-generation algorithm samples $\mathbf{r}_i \xleftarrow{\text{R}} \{0,1\}^{\hat{m}}$ and computes the verification vector $\mathbf{v}_i = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i)$. It then defines the public key $\mathbf{t}_i := \mathbf{B}\mathbf{r}_i + \mathbf{p} - \mathbf{A}\mathbf{v}_i \in \mathbb{Z}_q^n$ and the initial local opening $\widetilde{\mathbf{z}}_i := \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes \mathbf{t}_i, i)$, where $\mathbf{u}_i$ is the $i^{\text{th}}$ standard basis vector. It outputs the public key $\mathsf{pk}_i = \mathbf{t}_i$ and the secret key $\mathsf{sk}_i = (i, \mathbf{r}_i, \widetilde{\mathbf{z}}_i)$.
- $\mathsf{IncPK}(\mathsf{pp}, \mathsf{apk}, (i, \mathsf{pk}_i), \mathsf{op})$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, an aggregate public key $\mathsf{apk} = \widetilde{\mathbf{C}} \in \mathbb{Z}_q^{n\times m}$ (where we interpret $\mathsf{apk} = \perp$ to be the all-zeroes matrix), a public key $\mathsf{pk}_i = \mathbf{t}_i$ for user $i$, and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, the public-update algorithm computes the commitment

  $$\mathbf{C}_{(i)} = \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes \mathbf{t}_i). \tag{4.1}$$

  If $\mathsf{op} = \mathsf{Add}$, it outputs $\mathsf{apk}' = \widetilde{\mathbf{C}} + \mathbf{C}_{(i)}$. If $\mathsf{op} = \mathsf{Remove}$, it outputs $\mathsf{apk}' = \widetilde{\mathbf{C}} - \mathbf{C}_{(i)}$.
- $\mathsf{IncSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, (i, \mathsf{pk}_i), \mathsf{op})$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, an aggregate secret key $\mathsf{ask}_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*})$, a public key $\mathsf{pk}_i = \mathbf{t}_i$ for user $i$, and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, the private-update algorithm proceeds as follows:
  - If $i^* = i$, it returns $\mathsf{ask}_{i^*}$ unchanged. Otherwise, it computes the local opening

    $$\mathbf{z}_{(i,i^*)} = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes \mathbf{t}_i, i^*). \tag{4.2}$$

  - If $\mathsf{op} = \mathsf{Add}$, it outputs $\mathsf{ask}'_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*} + \mathbf{z}_{(i,i^*)})$.
  - If $\mathsf{op} = \mathsf{Remove}$, it outputs $\mathsf{ask}'_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*} - \mathbf{z}_{(i,i^*)})$.
- $\mathsf{Enc}(\mathsf{pp}, \mathsf{apk}, \mu)$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$ where $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$, an aggregate public key $\mathsf{apk} = \widetilde{\mathbf{C}}$, and a message $\mu \in \{0,1\}$, the encryption algorithm samples

  $$\mathbf{s} \xleftarrow{\text{R}} \mathbb{Z}_q^n, \quad \mathbf{e} \leftarrow \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\hat{m}}, \quad \mathbf{D}_1 \xleftarrow{\text{R}} \{0,1\}^{\hat{m}\times m}, \quad \mathbf{d}_2 \xleftarrow{\text{R}} \{0,1\}^{\hat{m}}.$$

  If $\|\mathbf{e}\| \ge \sqrt{\lambda}\,\widetilde{\sigma}$, it resets $\mathbf{e} = 0^{\hat{m}}$. It then outputs the ciphertext $\mathsf{ct} = (\mathbf{c}_1, \mathbf{c}_2, c_3)$ where

  $$\mathbf{c}_1^\intercal = \mathbf{s}^\intercal \mathbf{B} + \mathbf{e}^\intercal, \quad \mathbf{c}_2^\intercal = \mathbf{s}^\intercal(\mathbf{A} + \widetilde{\mathbf{C}}) + \mathbf{e}^\intercal \mathbf{D}_1, \quad c_3 = \mathbf{s}^\intercal \mathbf{p} + \mathbf{e}^\intercal \mathbf{d}_2 + \xi \cdot \mu.$$

- $\mathsf{Dec}(\mathsf{pp}, \mathsf{ask}_{i^*}, \mathsf{ct})$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, an aggregate secret key $\mathsf{ask}_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*})$, and a ciphertext $\mathsf{ct} = (\mathbf{c}_1, \mathbf{c}_2, c_3)$, the decryption algorithm computes the verification vector $\mathbf{v}_{i^*} = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i^*)$ and outputs

  $$\mu' = \lfloor c_3 + \mathbf{c}_1^\intercal(\mathbf{r}_{i^*} - \widetilde{\mathbf{z}}_{i^*}) - \mathbf{c}_2^\intercal \mathbf{v}_{i^*} \rceil_2.$$

**Theorem 4.12** (Correctness)**.** Suppose $n, m, \sigma, \widetilde{\sigma} = \mathsf{poly}(\lambda)$. Then, there is a fixed polynomial $\mathsf{poly}(\cdot)$ such that for all $q/\log q > 2^{\lambda} \cdot \mathsf{poly}(\lambda)$, Construction 4.11 is correct.

*Proof.* Fix a security parameter $\lambda$, a set of public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$ in the support of $\mathsf{Setup}(1^{\lambda})$, an index $i^* \in [2^{\lambda}]$, a key-pair $(\mathsf{pk}_{i^*}, \mathsf{sk}_{i^*})$ in the support of $\mathsf{KeyGen}(\mathsf{pp}, i^*)$, and a sequence of queries $\{(\mathsf{ind}_j, \mathsf{pk}_j, \mathsf{op}_j)\}_{j \in [Q]}$ where $Q \leq 2^{\lambda}$. For $j \in \{0, 1, \ldots, Q\}$, let $(\mathsf{apk}_j, \mathsf{ask}_{i^*,j}, S_j)$ be the values computed in Definition 4.4. Our goal is to show that if $S_Q[i^*] = \mathsf{pk}_{i^*}$, then for all messages $\mu \in \{0, 1\}$, we have $\mathsf{Dec}(\mathsf{pp}, \mathsf{ask}_{i^*,Q}, \mathsf{ct}) = \mu$ with probability 1, where $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{apk}_Q, \mu)$. First, we note the following:

- Since $(\mathsf{pk}_{i^*}, \mathsf{sk}_{i^*})$ is in the support of $\mathsf{KeyGen}(\mathsf{pp}, i^*)$, we can write $\mathsf{pk}_{i^*} = \mathbf{t}_{i^*} \in \mathbb{Z}_q^n$ and $\mathsf{sk}_{i^*} = (i^*, \mathbf{r}_{i^*}, \mathbf{z}_{(i^*,i^*)})$, where $\mathbf{r}_{i^*} \in \{0,1\}^{\hat{m}}$ and $\mathbf{z}_{(i^*,i^*)} \in \mathbb{Z}_q^{\hat{m}}$ is the local opening from Eq. (4.2), and

$$\mathbf{t}_{i^*} = \mathbf{B}\mathbf{r}_{i^*} + \mathbf{p} - \mathbf{A}\mathbf{v}_{i^*} \qquad \text{where} \qquad \mathbf{v}_{i^*} = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i^*). \tag{4.3}$$

- From Fact 3.5, we have $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$, where $\|\mathbf{R}\| \leq \sigma\sqrt{\lambda}$. Moreover, for all matrices $\mathbf{M} \in \mathbb{Z}_q^{n \times L}$ with $L \leq 2^{\lambda}$ and all indices $i \in [2^{\lambda}]$, the vectors $\mathbf{v}_i = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i)$ and $\mathbf{z}_i = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{M}, i)$ satisfy

$$\|\mathbf{v}_i\| \leq O\big(m^4 \sigma \log q \cdot \lambda^{1/2}\big) \qquad \text{and} \qquad \|\mathbf{z}_i\| \leq O\big(m^7 \sigma \log q \cdot \lambda^{3/2}\big). \tag{4.4}$$

**Induction invariant.** To prove the statement, we start by proving an induction invariant on the triples $(\mathsf{apk}_j, \mathsf{ask}_{i^*,j}, S_j)$. Specifically, for each $j \in \{0, 1, \ldots, Q\}$ and for each index $j' \in S_j$, let $\mathbf{t}_{j'} = S_j[j'] \in \mathbb{Z}_q^n$ be the public key associated with index $j'$. Let $\mathbf{C}_{(j')}$ and $\mathbf{z}_{(j',i^*)}$ be the commitment and local opening associated with $(j', \mathbf{t}_{j'})$ from Eqs. (4.1) and (4.2). We claim that the following two properties hold for all $j \in \{0, 1, \ldots, Q\}$:

- **Aggregated public key:** $\mathsf{apk}_j = \sum_{j' \in S_j} \mathbf{C}_{(j')}$, where we interpret $\mathsf{apk}_j = \perp$ as the all-zeroes matrix.
- **Aggregated secret key:** if $S_j[i^*] = \mathsf{pk}_{i^*}$, then $\mathsf{ask}_{i^*,j} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*,j})$ where $\widetilde{\mathbf{z}}_{i^*,j} = \sum_{j' \in S_j} \mathbf{z}_{(j',i^*)}$.

**Claim 4.13** (Invariant Preservation)**.** For all $j \in \{0, 1, \ldots, Q\}$, the induction invariant holds.

*Proof.* We proceed by induction on $j$.

**Base case.** For the base case where $j = 0$, we have $\mathsf{apk}_0 = \perp$, $\mathsf{ask}_{i^*,0} = \perp$, and $S_0 = \varnothing$. Both properties hold trivially since $S_0 = \varnothing$.

**Inductive step.** For the inductive step, suppose the invariant holds for $(\mathsf{apk}_{j-1}, \mathsf{ask}_{i^*,j-1}, S_{j-1})$ and consider the query $(\mathsf{ind}_j, \mathsf{pk}_j, \mathsf{op}_j)$. To simplify notation, let $i = \mathsf{ind}_j$. First, if neither of the conditions ($\mathsf{op}_j = \mathsf{Add}$ and $i \notin S_{j-1}$) or ($\mathsf{op}_j = \mathsf{Remove}$ and $S_{j-1}[i] = \mathsf{pk}_j$) holds, then UpdateAPK and UpdateASK leave their inputs unchanged. In this case,

$$\mathsf{apk}_j = \mathsf{apk}_{j-1} \quad \text{and} \quad \mathsf{ask}_{i^*,j} = \mathsf{ask}_{i^*,j-1} \quad \text{and} \quad S_j = S_{j-1}.$$

Thus, if the induction invariant holds for $(\mathsf{apk}_{j-1}, \mathsf{ask}_{i^*,j-1}, S_{j-1})$, then it also holds for $(\mathsf{apk}_j, \mathsf{ask}_{i^*,j}, S_j)$. It remains to consider the two cases where the query is non-trivial:

- **Case** $\mathsf{op}_j = \mathsf{Add}$ **and** $i \notin S_{j-1}$**:** In this case, $S_j = S_{j-1} \cup \{i\}$ with $S_j[i] = \mathsf{pk}_j$ and $S_j[j'] = S_{j-1}[j']$ for all $j' \in S_{j-1}$. By construction, IncPK outputs

$$\mathsf{apk}_j = \mathsf{apk}_{j-1} + \mathbf{C}_{(i)} = \sum_{j' \in S_{j-1}} \mathbf{C}_{(j')} + \mathbf{C}_{(i)} = \sum_{j' \in S_j} \mathbf{C}_{(j')},$$

  where the second equality uses the inductive hypothesis. For the aggregated secret key, suppose $S_j[i^*] = \mathsf{pk}_{i^*}$ (otherwise, the second property holds vacuously). We consider two sub-cases:

– Suppose $i^* \neq i$. Then $S_{j-1}[i^*] = S_j[i^*] = \mathsf{pk}_{i^*}$. Then, by the inductive hypothesis, we have that $\mathsf{ask}_{i^*,j-1} = (i^*, \mathbf{r}_{i^*}, \sum_{j' \in S_{j-1}} \mathbf{z}_{(j',i^*)})$. In this case, UpdateASK invokes IncSK on $(i, \mathsf{pk}_j)$ with operation Add, which adds the local opening $\mathbf{z}_{(i,i^*)}$ to the aggregated opening. Thus,

$$\widetilde{\mathbf{z}}_{i^*,j} = \sum_{j' \in S_{j-1}} \mathbf{z}_{(j',i^*)} + \mathbf{z}_{(i,i^*)} = \sum_{j' \in S_j} \mathbf{z}_{(j',i^*)}.$$

– Suppose $i^* = i$. In this case, $\mathsf{pk}_j = S_j[i^*] = \mathsf{pk}_{i^*}$. Then UpdateASK initializes $\mathsf{ask}_{i^*} = \mathsf{sk}_{i^*} = (i^*, \mathbf{r}_{i^*}, \mathbf{z}_{(i^*,i^*)})$ and invokes IncSK on $(j', \mathbf{t}_{j'})$ with operation Add for each member $j' \in S_{j-1}$, which adds the local opening $\mathbf{z}_{(j',i^*)}$. Thus,

$$\widetilde{\mathbf{z}}_{i^*,j} = \mathbf{z}_{(i^*,i^*)} + \sum_{j' \in S_{j-1}} \mathbf{z}_{(j',i^*)} = \sum_{j' \in S_j} \mathbf{z}_{(j',i^*)}.$$

In both cases, the induction invariant holds.

- **Case** $\mathsf{op}_j = \mathsf{Remove}$ **and** $S_{j-1}[i] = \mathsf{pk}_j$**:** In this case, $S_j = S_{j-1} \setminus \{i\}$. Let $\mathsf{pk}_j = \mathbf{t}_i$. By construction, IncPK first computes $\widetilde{\mathbf{C}}_{(i)} = \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes \mathbf{t}_i)$. Since $S_{j-1}[i] = \mathsf{pk}_j = \mathbf{t}_i$, this means $\mathbf{C}_{(i)} = \widetilde{\mathbf{C}}_{(i)}$. Thus, the output of IncPK satisfies

$$\mathsf{apk}_j = \mathsf{apk}_{j-1} - \widetilde{\mathbf{C}}_{(i)} = \sum_{j' \in S_{j-1}} \mathbf{C}_{(j')} - \mathbf{C}_{(i)} = \sum_{j' \in S_j} \mathbf{C}_{(j')}.$$

For the aggregated secret key, suppose $S_j[i^*] = \mathsf{pk}_{i^*}$ (otherwise the second property holds vacuously). This means that $i^* \neq i$ and $S_{j-1}[i^*] = \mathsf{pk}_{i^*}$, so by the inductive hypothesis, $\mathsf{ask}_{i^*,j-1} = (i^*, \mathbf{r}_{i^*}, \sum_{j' \in S_{j-1}} \mathbf{z}_{(j',i^*)})$. In this case, UpdateASK invokes IncSK on $(i, \mathsf{pk}_j)$ with operation Remove, which subtracts the local opening $\mathbf{z}_{(i,i^*)}$. Since Open is deterministic, this means

$$\widetilde{\mathbf{z}}_{i^*,j} = \sum_{j' \in S_{j-1}} \mathbf{z}_{(j',i^*)} - \mathbf{z}_{(i,i^*)} = \sum_{j' \in S_j} \mathbf{z}_{(j',i^*)}.$$

In both cases, we again see that the induction invariant holds. The claim now follows by induction. □

**Completing the proof.** To complete the proof, suppose $S_Q[i^*] = \mathsf{pk}_{i^*}$, and write $\widetilde{\mathbf{C}} = \mathsf{apk}_Q$ and $\mathsf{ask}_{i^*,Q} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*,Q})$. By Claim 4.13, this means

$$\widetilde{\mathbf{C}} = \sum_{j' \in S_Q} \mathbf{C}_{(j')} \qquad \text{and} \qquad \widetilde{\mathbf{z}}_{i^*,Q} = \sum_{j' \in S_Q} \mathbf{z}_{(j',i^*)}.$$

From Eqs. (4.1) and (4.2), we have that $\mathbf{C}_{(j')} = \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{j'}^\intercal \otimes \mathbf{t}_{j'})$ and $\mathbf{z}_{(j',i^*)} = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{j'}^\intercal \otimes \mathbf{t}_{j'}, i^*)$. By Claim 4.10, this means

$$\widetilde{\mathbf{C}}\mathbf{v}_{i^*} = \sum_{j' \in S_Q} \mathbf{C}_{(j')} \cdot \mathbf{v}_{i^*} = \mathbf{t}_{i^*} - \mathbf{B} \sum_{j' \in S_Q} \mathbf{z}_{(j',i^*)} = \mathbf{t}_{i^*} - \mathbf{B}\widetilde{\mathbf{z}}_{i^*,Q}. \tag{4.5}$$

Moreover, by Eq. (4.4),

$$\|\widetilde{\mathbf{z}}_{i^*,Q}\| \leq |S_Q| \cdot O\big(m^7 \sigma \log q \cdot \lambda^{3/2}\big). \tag{4.6}$$

Now take any message $\mu \in \{0,1\}$ and let $\mathsf{ct} = (\mathbf{c}_1, \mathbf{c}_2, c_3)$ be an encryption of $\mu$. By construction of Construction 4.11, we can write

$$\mathbf{c}_1^\intercal = \mathbf{s}^\intercal \mathbf{B} + \mathbf{e}^\intercal, \quad \mathbf{c}_2^\intercal = \mathbf{s}^\intercal(\mathbf{A} + \widetilde{\mathbf{C}}) + \mathbf{e}^\intercal \mathbf{D}_1, \quad c_3 = \mathbf{s}^\intercal \mathbf{p} + \mathbf{e}^\intercal \mathbf{d}_2 + \xi\mu.$$

The decryption algorithm $\mathsf{Dec}(\mathsf{pp}, \mathsf{ask}_{i^*,Q}, \mathsf{ct})$ now computes

$$\mu' = \lfloor c_3 + \mathbf{c}_1^\intercal(\mathbf{r}_{i^*} - \widetilde{\mathbf{z}}_{i^*,Q}) - \mathbf{c}_2^\intercal \mathbf{v}_{i^*} \rceil_2.$$

Substituting the ciphertext components and regrouping terms gives

$$\mu' = \lfloor \xi\mu + \mathbf{e}^\intercal \mathbf{d}_2 + \mathbf{e}^\intercal \mathbf{r}_{i^*} - \mathbf{e}^\intercal \mathbf{D}_1 \mathbf{v}_{i^*} - \mathbf{e}^\intercal \widetilde{\mathbf{z}}_{i^*,Q} + \mathbf{s}^\intercal\big(\mathbf{p} + \mathbf{B}\mathbf{r}_{i^*} - \mathbf{B}\widetilde{\mathbf{z}}_{i^*,Q} - \mathbf{A}\mathbf{v}_{i^*} - \widetilde{\mathbf{C}}\mathbf{v}_{i^*}\big) \rceil_2.$$

By Eq. (4.3), $\mathbf{p} + \mathbf{B}\mathbf{r}_{i^*} - \mathbf{A}\mathbf{v}_{i^*} = \mathbf{t}_{i^*}$, and by Eq. (4.5), $\widetilde{\mathbf{C}}\mathbf{v}_{i^*} = \mathbf{t}_{i^*} - \mathbf{B}\widetilde{\mathbf{z}}_{i^*,Q}$, so we have

$$\mu' = \lfloor \xi\mu + \underbrace{\mathbf{e}^\mathsf{T}\mathbf{d}_2 + \mathbf{e}^\mathsf{T}\mathbf{r}_{i^*} - \mathbf{e}^\mathsf{T}\mathbf{D}_1\mathbf{v}_{i^*} - \mathbf{e}^\mathsf{T}\widetilde{\mathbf{z}}_{i^*,Q}}_{\nu} \rceil_2.$$

It remains to bound $|\nu|$. By construction of Enc, $\|\mathbf{e}\| \le \lambda^{1/2}\widetilde{\sigma}$. In addition, $\mathbf{r}_{i^*}, \mathbf{d}_2 \in \{0,1\}^{\hat{m}}$, $\mathbf{D}_1 \in \{0,1\}^{\hat{m}\times m}$. Thus, by Eq. (4.4) and Eq. (4.6), this gives

$$|\nu| \le \hat{m}\,\lambda^{1/2}\widetilde{\sigma}\left(2 + O\big(m^5\sigma\log q\cdot\lambda^{1/2}\big) + |S_Q|\cdot O\big(m^7\sigma\log q\cdot\lambda^{3/2}\big)\right) \le |S_Q|\cdot O\big(\hat{m}m^7\sigma\widetilde{\sigma}\cdot\lambda^2\log q\big),$$

where the second inequality uses $|S_Q| \ge 1$ (which holds since $i^* \in S_Q$). Finally, since $|S_Q| \le Q \le 2^\lambda$ and $n, m, \sigma, \widetilde{\sigma} = \mathsf{poly}(\lambda)$, we conclude that

$$|\nu| \le 2^\lambda \cdot \mathsf{poly}(\lambda)\cdot\log q.$$

Setting $q/\log q > 4\cdot 2^\lambda\cdot\mathsf{poly}(\lambda)$ means $|\nu| < q/4$. Since $\xi = \lfloor q/2\rfloor$, the rounding operation recovers $\mu' = \mu$ and correctness holds. □

**Theorem 4.14** (Succinctness)**.** Suppose $n, m = \mathsf{poly}(\lambda)$ and $q \le 2^{\mathsf{poly}(\lambda)}$. Then, Construction 4.11 satisfies succinctness (Definition 4.5).

*Proof.* Take any $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$ in the support of $\mathsf{Setup}(1^\lambda)$. By Definition 4.5, it suffices to bound the size of the aggregated public key output by $\mathsf{IncPK}(\mathsf{pp},\cdot,\cdot,\cdot)$ and the size of the aggregated secret key output by $\mathsf{IncSK}(\mathsf{pp},\cdot,\cdot,\cdot)$ by a universal polynomial in $\lambda$:

- Algorithm IncPK outputs a matrix $\mathsf{apk}' \in \mathbb{Z}_q^{n\times m}$, which can be represented using $nm\lceil\log q\rceil$ bits.
- Algorithm IncSK outputs a triple $\mathsf{ask}'_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*})$, where $i^* \in [2^\lambda]$ and $\mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*} \in \mathbb{Z}_q^{\hat{m}}$. These can be represented using $\lambda + 2\hat{m}\lceil\log q\rceil$ bits.

When $n, m = \mathsf{poly}(\lambda)$, $\hat{m} = 4m^5$, and $q \le 2^{\mathsf{poly}(\lambda)}$, these outputs have size $\mathsf{poly}(\lambda)$, as required. □

**Theorem 4.15** (Static Security)**.** If the matrix commitment scheme is secure (Fact 3.5), $n \ge \lambda$, $m \ge 2(n+1)\log q$, $\sigma \ge \log m$, $m = \mathsf{poly}(\lambda)$, and $q$ is prime, the construction above satisfies static security (Definition 4.9).

*Proof.* To prove security, we proceed by defining a sequence of hybrid experiments, each parameterized by a bit $b \in \{0,1\}$ and implicitly by an adversary $\mathcal{A}$ and a security parameter $\lambda$:

- $\mathsf{Hyb}_0^{(b)}$: This is the static security experiment with challenge bit $b \in \{0,1\}$. Recall that in the static security experiment, the adversary commits to all of its pre-challenge queries at the beginning of the experiment, before seeing the scheme parameters. Concretely, the experiment proceeds as follows:
  - **Static commitment phase:** At the beginning of the experiment, the adversary $\mathcal{A}$ commits to the entire sequence of pre-challenge queries it will make, where each query is one of the following:
    - a key-generation query on an index $i \in [2^\lambda]$;
    - a corruption query on a counter $c$; or
    - an update query on a counter $c$ and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$.
  - **Setup phase:** The challenger begins by sampling the public parameters

    $$\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B}) \leftarrow \mathsf{Setup}_{\mathsf{com}}(1^\lambda), \qquad \mathbf{A} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n\times m}, \qquad \mathbf{p} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n.$$

    Next, the challenger initializes a counter $\mathsf{ctr} = 0$, a dictionary $\mathsf{D} = \varnothing$, a mapping $S = \varnothing$, a set $T = \varnothing$, the aggregated public key $\widetilde{\mathbf{C}} = \mathbf{0}^{n\times m}$ (representing $\mathsf{apk} = \perp$), and a set of corrupted counters $C = \varnothing$. The challenger processes each of the committed queries in order and constructs the response to each query as follows:

* **Key-generation query on an index $i \in [2^\lambda]$:** The challenger increments the counter $\mathsf{ctr} = \mathsf{ctr} + 1$. It samples $\mathbf{r}_{\mathsf{ctr}} \xleftarrow{R} \{0,1\}^{\hat{m}}$ and computes

$$\mathbf{v}_i = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i) \in \mathbb{Z}_q^m \qquad \text{and} \qquad \mathbf{t}_{\mathsf{ctr}} = \mathbf{B}\mathbf{r}_{\mathsf{ctr}} + \mathbf{p} - \mathbf{A}\mathbf{v}_i \in \mathbb{Z}_q^n,$$

together with the local opening $\mathbf{z}_{\mathsf{ctr}} = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes \mathbf{t}_{\mathsf{ctr}}, i)$. The challenger sets $\mathsf{pk}_{\mathsf{ctr}} = \mathbf{t}_{\mathsf{ctr}}$, $\mathsf{sk}_{\mathsf{ctr}} = (i, \mathbf{r}_{\mathsf{ctr}}, \mathbf{z}_{\mathsf{ctr}})$, and $\mathsf{ask}_{\mathsf{ctr}} = \bot$, adds the mapping $\mathsf{ctr} \mapsto (i, \mathsf{pk}_{\mathsf{ctr}}, \mathsf{sk}_{\mathsf{ctr}}, \mathsf{ask}_{\mathsf{ctr}})$ to D, and defines the response to the query to be $\mathsf{pk}_{\mathsf{ctr}}$.
* **Corruption query on a counter $c \in [\mathsf{ctr}]$:** The challenger looks up $(i, \mathsf{pk}_c, \mathsf{sk}_c, \mathsf{ask}_c) = \mathsf{D}[c]$ and defines the response to the query to be $(\mathsf{sk}_c, \mathsf{ask}_c)$. In addition, the challenger updates $C = C \cup \{c\}$.
* **Update query on a counter $c \in [\mathsf{ctr}]$ and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$:** The challenger looks up $(i, \mathsf{pk}_c, \mathsf{sk}_c, \mathsf{ask}_c) = \mathsf{D}[c]$ and writes $\mathsf{pk}_c = \mathbf{t}_c$ and $\mathsf{sk}_c = (i, \mathbf{r}_c, \mathbf{z}_c)$. If either ($\mathsf{op} = \mathsf{Add}$ and $i \in S$) or ($\mathsf{op} = \mathsf{Remove}$ and $c \notin T$), the challenger sets the response to this query to be $\bot$. Otherwise, the challenger computes the commitment

$$\mathbf{C}_{(c)} = \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes \mathbf{t}_c) \in \mathbb{Z}_q^{n \times m}.$$

In the following, for counters $\tau, \tau' \in T \cup \{c\}$, we write $(i_\tau, \mathbf{t}_\tau)$ to denote the index and public key associated with $\mathsf{D}[\tau]$, and we write

$$\mathbf{z}_{(\tau,\tau')} = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_\tau}^\intercal \otimes \mathbf{t}_\tau, i_{\tau'})$$

to denote the respective local openings. The challenger then proceeds based on the operation:

  · **Case** $\mathsf{op} = \mathsf{Add}$**:** For each $\tau \in T$, the challenger looks up $(i_\tau, \mathsf{pk}_\tau, \mathsf{sk}_\tau, \mathsf{ask}_\tau) = \mathsf{D}[\tau]$, parses $\mathsf{ask}_\tau = (i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau)$, and updates $\mathsf{ask}_\tau$ in $\mathsf{D}[\tau]$ with $(i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau + \mathbf{z}_{(c,\tau)})$. For the newly added user, the challenger computes

$$\mathsf{ask}_c = \left(i, \mathbf{r}_c, \mathbf{z}_c + \sum_{\tau \in T} \mathbf{z}_{(\tau,c)}\right)$$

and updates $\mathsf{ask}_c$ in $\mathsf{D}[c]$. Finally, the challenger updates the aggregated public key $\widetilde{\mathbf{C}} = \widetilde{\mathbf{C}} + \mathbf{C}_{(c)}$, sets $S[i] = \mathbf{t}_c$, and updates $T = T \cup \{c\}$.

  · **Case** $\mathsf{op} = \mathsf{Remove}$**:** For each $\tau \in T \setminus \{c\}$, the challenger looks up $(i_\tau, \mathsf{pk}_\tau, \mathsf{sk}_\tau, \mathsf{ask}_\tau) = \mathsf{D}[\tau]$, parses $\mathsf{ask}_\tau = (i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau)$, and updates $\mathsf{ask}_\tau = (i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau - \mathbf{z}_{(c,\tau)})$ in $\mathsf{D}[\tau]$. It also sets $\mathsf{ask}_c = \bot$ in $\mathsf{D}[c]$. Finally, the challenger updates the aggregated public key $\widetilde{\mathbf{C}} = \widetilde{\mathbf{C}} - \mathbf{C}_{(c)}$, removes $i$ from $S$, and updates $T = T \setminus \{c\}$.

The challenger sets the response to the query to be $\left(\widetilde{\mathbf{C}}, \{(\tau, \mathsf{ask}_\tau)\}_{\tau \in T \cap C}\right)$.

After computing the responses to all of the committed pre-challenge queries, the challenger constructs the challenge ciphertext as follows:

* If $T = \emptyset$ or $T \cap C \neq \emptyset$, the challenger halts the experiment with output 0.
* Otherwise, the challenger sets $\hat{T} = T$ and $\hat{\mathbf{C}} = \widetilde{\mathbf{C}}$. It samples

$$\hat{\mathbf{s}} \xleftarrow{R} \mathbb{Z}_q^n, \qquad \hat{\mathbf{e}} \leftarrow \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\hat{m}}, \qquad \hat{\mathbf{D}}_1 \xleftarrow{R} \{0,1\}^{\hat{m} \times m}, \qquad \hat{\mathbf{d}}_2 \xleftarrow{R} \{0,1\}^{\hat{m}}.$$

If $\|\hat{\mathbf{e}}\| \geq \sqrt{\lambda}\,\widetilde{\sigma}$, it resets $\hat{\mathbf{e}} = \mathbf{0}^{\hat{m}}$. The challenger then constructs the challenge ciphertext $\mathsf{ct}^* = (\hat{\mathbf{c}}_1, \hat{\mathbf{c}}_2, \hat{c}_3)$ as follows:

$$\begin{aligned}
\hat{\mathbf{c}}_1^\intercal &= \hat{\mathbf{s}}^\intercal \mathbf{B} + \hat{\mathbf{e}}^\intercal \\
\hat{\mathbf{c}}_2^\intercal &= \hat{\mathbf{s}}^\intercal (\mathbf{A} + \hat{\mathbf{C}}) + \hat{\mathbf{e}}^\intercal \hat{\mathbf{D}}_1 \\
\hat{c}_3 &= \hat{\mathbf{s}}^\intercal \mathbf{p} + \hat{\mathbf{e}}^\intercal \hat{\mathbf{d}}_2 + \xi \cdot b.
\end{aligned}$$

Finally, the challenger gives the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, the responses to each of the committed pre-challenge queries, and the challenge ciphertext $\mathsf{ct}^*$ to $\mathcal{A}$.

- **Post-challenge query phase:** The adversary may now make additional (adaptive) key-generation, corruption, and update queries. The challenger computes the response to each query exactly as it processed the corresponding pre-challenge queries. The adversary is not allowed to make corruption queries on counters $c \in \hat{T}$ (i.e., on public keys that are associated with the challenge ciphertext).
- **Output phase:** At the end of the experiment, the adversary outputs a bit $b' \in \{0,1\}$, which is the output of the experiment.

We write $\mathsf{Hyb}_0^{(b)}(\mathcal{A})$ to denote the output of an execution of $\mathsf{Hyb}_0^{(b)}$ with adversary $\mathcal{A}$ and the challenge bit $b \in \{0,1\}$.

- $\mathsf{Hyb}_1^{(b)}$: Same as $\mathsf{Hyb}_0^{(b)}$, except the challenger samples the public keys associated with the counters in the challenge set $\hat{T}$ uniformly at random. Recall that in the static security experiment, the adversary commits to all of its pre-challenge queries at the beginning of the experiment, so the set $\hat{T}$ of counters associated with the challenge ciphertext is fully determined by the adversary's committed query sequence. Specifically, the challenger implements the following changes to the setup phase:
  - At the beginning of the setup phase, the challenger computes the sets $\hat{T}$ and $C$ determined by the committed sequence of queries; if $\hat{T} = \emptyset$ or $\hat{T} \cap C \neq \emptyset$, the challenger immediately halts the experiment with output 0.
  - When responding to a key-generation query whose counter satisfies $\mathsf{ctr} \in \hat{T}$, the challenger now samples $\mathbf{t}_{\mathsf{ctr}} \xleftarrow{R} \mathbb{Z}_q^n$ instead of setting $\mathbf{t}_{\mathsf{ctr}} = \mathbf{B}\mathbf{r}_{\mathsf{ctr}} + \mathbf{p} - \mathbf{A}\mathbf{v}_i$. In this case, the challenger also sets $\mathsf{sk}_{\mathsf{ctr}} = \perp$ and $\mathsf{ask}_{\mathsf{ctr}} = \perp$ in D. When implementing update queries, the challenger does not update $\mathsf{ask}_{\mathsf{ctr}}$.

  Note that this experiment remains well-defined: the values $\mathsf{sk}_c$ and $\mathsf{ask}_c$ for a counter $c \in \hat{T}$ are only needed if the adversary makes a corruption query on $c$ or makes an update query and $c \in T \cap C$. In both cases, it will be the case that $c \in C$ and so $C \cap \hat{T} \neq \emptyset$. In this case, the challenger always outputs 0.
- $\mathsf{Hyb}_2^{(b)}$: Same as $\mathsf{Hyb}_1^{(b)}$, except the challenger programs the commitment to the challenge set into the public parameters. For the set $\hat{T}$ determined by the committed sequence of queries, the challenger samples the public keys $\mathbf{t}_c \xleftarrow{R} \mathbb{Z}_q^n$ for all $c \in \hat{T}$ (as in $\mathsf{Hyb}_1^{(b)}$), together with $\hat{\mathbf{D}}_1 \xleftarrow{R} \{0,1\}^{\hat{m} \times m}$ and $\hat{\mathbf{d}}_2 \xleftarrow{R} \{0,1\}^{\hat{m}}$. It then computes the aggregated public key associated with the challenge ciphertext

$$\hat{\mathbf{C}} = \sum_{c \in \hat{T}} \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_c}^{\mathsf{T}} \otimes \mathbf{t}_c),$$

  where $i_c$ denotes the index associated with counter $c$ in $\mathsf{D}[c]$. Then the challenger sets

$$\mathbf{A} = \mathbf{B}\hat{\mathbf{D}}_1 - \hat{\mathbf{C}} \qquad \text{and} \qquad \mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_2.$$

  The rest of the experiment is unchanged. In particular, the challenger uses the values $\hat{\mathbf{D}}_1, \hat{\mathbf{d}}_2$ sampled above when constructing the challenge ciphertext. Note that this is again well-defined: in $\mathsf{Hyb}_1^{(b)}$, the public keys $\mathbf{t}_c$ for $c \in \hat{T}$ are sampled independently of $\mathbf{A}$ and $\mathbf{p}$, so the challenger can compute $\hat{\mathbf{C}}$ before defining $\mathbf{A}$ and $\mathbf{p}$.
- $\mathsf{Hyb}_3^{(b)}$: Same as $\mathsf{Hyb}_2^{(b)}$, except the challenger does not truncate $\hat{\mathbf{e}}$ when constructing the challenge ciphertext (i.e., it no longer resets $\hat{\mathbf{e}} = \mathbf{0}^{\hat{m}}$ when $\|\hat{\mathbf{e}}\| \geq \sqrt{\lambda}\,\widetilde{\sigma}$).
- $\mathsf{Hyb}_4^{(b)}$: Same as $\mathsf{Hyb}_3^{(b)}$, except when constructing the challenge ciphertext, the challenger samples $\hat{\mathbf{c}}_1 \xleftarrow{R} \mathbb{Z}_q^{\hat{m}}$. It computes the remaining components of the challenge ciphertext as

$$\hat{\mathbf{c}}_2^{\mathsf{T}} = \hat{\mathbf{c}}_1^{\mathsf{T}}\hat{\mathbf{D}}_1 \qquad \text{and} \qquad \hat{c}_3 = \hat{\mathbf{c}}_1^{\mathsf{T}}\hat{\mathbf{d}}_2 + \xi \cdot b.$$

- $\mathsf{Hyb}_5^{(b)}$: Same as $\mathsf{Hyb}_4^{(b)}$, except the challenger samples $\mathbf{p} \xleftarrow{R} \mathbb{Z}_q^n$ (instead of setting $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_2$) and $\hat{c}_3 \xleftarrow{R} \mathbb{Z}_q$ (instead of setting $\hat{c}_3 = \hat{\mathbf{c}}_1^{\mathsf{T}}\hat{\mathbf{d}}_2 + \xi \cdot b$). In this experiment, the challenger's behavior is independent of the challenge bit $b$.

We write $\mathsf{Hyb}_i^{(b)}(\mathcal{A})$ to denote the output of an execution of $\mathsf{Hyb}_i^{(b)}$ with adversary $\mathcal{A}$. We now analyze each pair of adjacent distributions.

**Claim 4.16.** Suppose $n \geq \lambda$, $m \geq 2(n+1)\log q$, $\sigma \geq \log m$, and $q$ is prime. Then, for all efficient adversaries $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_0^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_1^{(b)}(\mathcal{A})$ are statistically indistinguishable.

*Proof.* We first argue that the two syntactic differences between the experiments do not affect the output distribution:

- **Admissibility check:** In $\mathsf{Hyb}_1^{(b)}$, the challenger performs the check that $\hat{T} = \varnothing$ or $\hat{T} \cap C \neq \varnothing$ at the beginning of the setup phase rather than after processing the committed queries. Since the sets $\hat{T}$ and $C$ are fully determined by the committed sequence of queries (and *not* by the values sampled by the challenger when responding to the queries), the outcome of the check is identically distributed in the two experiments.
- **Secret keys for the counters in $\hat{T}$:** In $\mathsf{Hyb}_1^{(b)}$, the challenger sets $\mathsf{sk}_c = \mathsf{ask}_c = \perp$ for the counters $c \in \hat{T}$. Consider an execution that does not halt with output 0. In this case, no counter in $\hat{T}$ is corrupted (i.e., $\hat{T} \cap C = \varnothing$), so the values $\mathsf{sk}_c$ and $\mathsf{ask}_c$ for $c \in \hat{T}$ do not appear in the response to any corruption query, nor in the response to any update query (which contains $\mathsf{ask}_\tau$ only for the corrupted counters $\tau \in T \cap C$). Thus, the responses sent to the adversary are identical in the two experiments.

It thus suffices to analyze the distribution of the public keys $\mathbf{t}_c \in \mathbb{Z}_q^n$ for the counters $c \in \hat{T}$:

- In $\mathsf{Hyb}_0^{(b)}$, the challenger sets $\mathbf{t}_c = \mathbf{B}\mathbf{r}_c + \mathbf{p} - \mathbf{A}\mathbf{v}_{i_c}$ where $\mathbf{r}_c \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$. Nothing else in this experiment depends on the randomness $\mathbf{r}_c$.
- In $\mathsf{Hyb}_1^{(b)}$, the challenger samples $\mathbf{t}_c \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$.

By Lemma 3.4, over the choice of $\mathbf{r}_c \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$, the distributions

$$(\mathbf{W},\ \mathbf{R},\ \mathbf{B}\mathbf{r}_c) \qquad \text{and} \qquad (\mathbf{W},\ \mathbf{R},\ \mathbf{t}_c)$$

are statistically close, where $\mathbf{t}_c \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$ and $\mathbf{B}$ is the matrix determined by $(\mathbf{W}, \mathbf{R})$ in $\mathsf{pp}_{\mathsf{com}}$. Since the vector $\mathbf{v}_{i_c} = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i_c)$ is a deterministic function of $\mathsf{pp}_{\mathsf{com}}$ and the parameters $(\mathbf{A}, \mathbf{p})$ are sampled independently of $\mathbf{r}_c$, this means the distribution of $\mathbf{t}_c = \mathbf{B}\mathbf{r}_c + \mathbf{p} - \mathbf{A}\mathbf{v}_{i_c}$ in $\mathsf{Hyb}_0^{(b)}$ is statistically close to uniform over $\mathbb{Z}_q^n$ (and independent of all other quantities). This coincides with the distribution of $\mathbf{t}_c$ in $\mathsf{Hyb}_1^{(b)}$. Since $\mathcal{A}$ is efficient, it makes at most a polynomial number of key-generation queries. The claim now follows by a standard hybrid argument. □

**Claim 4.17.** Suppose $n \geq \lambda$, $m \geq 2(n+1)\log q$, $\sigma \geq \log m$, and $q$ is prime. Then, for all adversaries $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_1^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_2^{(b)}(\mathcal{A})$ are statistically indistinguishable.

*Proof.* The only difference between the two experiments is the distribution of the public parameters $(\mathbf{A}, \mathbf{p})$:

- In $\mathsf{Hyb}_1^{(b)}$, the challenger samples $\mathbf{A} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n \times m}$ and $\mathbf{p} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$.
- In $\mathsf{Hyb}_2^{(b)}$, the challenger sets $\mathbf{A} = \mathbf{B}\hat{\mathbf{D}}_1 - \hat{\mathbf{C}}$ and $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_2$, where $\hat{\mathbf{D}}_1 \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m} \times m}$ and $\hat{\mathbf{d}}_2 \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$ are the randomness of the challenge ciphertext and $\hat{\mathbf{C}} = \sum_{c \in \hat{T}} \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_c}^{\mathsf{T}} \otimes \mathbf{t}_c)$.

The values $(\hat{\mathbf{D}}_1, \hat{\mathbf{d}}_2)$ are sampled independently of everything else in the experiments. Moreover, other than in the definition of $(\mathbf{A}, \mathbf{p})$ in $\mathsf{Hyb}_2^{(b)}$, the values $(\hat{\mathbf{D}}_1, \hat{\mathbf{d}}_2)$ only appear via the terms $\hat{\mathbf{e}}^{\mathsf{T}}\hat{\mathbf{D}}_1$ and $\hat{\mathbf{e}}^{\mathsf{T}}\hat{\mathbf{d}}_2$ in the challenge ciphertext components

$$\hat{\mathbf{c}}_2^{\mathsf{T}} = \hat{\mathbf{s}}^{\mathsf{T}}(\mathbf{A} + \hat{\mathbf{C}}) + \hat{\mathbf{e}}^{\mathsf{T}}\hat{\mathbf{D}}_1 \qquad \text{and} \qquad \hat{c}_3 = \hat{\mathbf{s}}^{\mathsf{T}}\mathbf{p} + \hat{\mathbf{e}}^{\mathsf{T}}\hat{\mathbf{d}}_2 + \xi \cdot b.$$

Finally, the matrix $\hat{\mathbf{C}}$ is determined by the committed sequence of queries together with the public keys $\mathbf{t}_c \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$ for $c \in \hat{T}$. All of these values are sampled independently of $(\hat{\mathbf{D}}_1, \hat{\mathbf{d}}_2)$ and of $(\mathbf{A}, \mathbf{p})$. By Lemma 3.4, the distributions

$$\left(\mathbf{W},\ \mathbf{R},\ \mathbf{B}[\hat{\mathbf{D}}_1 \mid \hat{\mathbf{d}}_2],\ \hat{\mathbf{e}}^{\mathsf{T}}[\hat{\mathbf{D}}_1 \mid \hat{\mathbf{d}}_2]\right) \qquad \text{and} \qquad \left(\mathbf{W},\ \mathbf{R},\ [\tilde{\mathbf{A}} \mid \tilde{\mathbf{p}}],\ \hat{\mathbf{e}}^{\mathsf{T}}[\hat{\mathbf{D}}_1 \mid \hat{\mathbf{d}}_2]\right)$$

are statistically close, where $\tilde{\mathbf{A}} \xleftarrow{R} \mathbb{Z}_q^{n\times m}$ and $\tilde{\mathbf{p}} \xleftarrow{R} \mathbb{Z}_q^n$. The distribution on the left maps to $\mathsf{Hyb}_2^{(b)}$ when we set $\mathbf{A} = \mathbf{B}\hat{\mathbf{D}}_1 - \hat{\mathbf{C}}$ and $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_2$, whereas the distribution on the right maps to $\mathsf{Hyb}_1^{(b)}$ when we set $\mathbf{A} = \tilde{\mathbf{A}} - \hat{\mathbf{C}}$ and $\mathbf{p} = \tilde{\mathbf{p}}$ (specifically, since $\tilde{\mathbf{A}}$ is uniform and independent of $\hat{\mathbf{C}}$, the matrix $\tilde{\mathbf{A}} - \hat{\mathbf{C}}$ is also uniform).

To conclude, we observe that the remainder of the two experiments is the *same* deterministic function of the quantities shown above along with coins that are independent of $(\hat{\mathbf{D}}_1, \hat{\mathbf{d}}_2)$. Concretely, the public parameters are $\mathbf{A} = \mathbf{B}\hat{\mathbf{D}}_1 - \hat{\mathbf{C}}$ and $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_2$ (respectively $\tilde{\mathbf{A}} - \hat{\mathbf{C}}$ and $\tilde{\mathbf{p}}$, which are uniform since $\hat{\mathbf{C}}$ is independent). Every public key of a non-challenge user, the aggregated keys, and the challenge ciphertext are derived from $(\mathbf{A}, \mathbf{p})$ and independent coins. The components above are computed from $\mathbf{A}$, $\mathbf{p}$, $\hat{\mathbf{s}}$ and $\hat{\mathbf{e}}^\intercal[\hat{\mathbf{D}}_1 \mid \hat{\mathbf{d}}_2]$. Statistical closeness of the two experiments therefore follows. □

**Claim 4.18.** Suppose $m = \mathsf{poly}(\lambda)$. Then, for all adversaries $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_2^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_3^{(b)}(\mathcal{A})$ are statistically indistinguishable.

*Proof.* The only difference between the two experiments is the truncation of the error vector $\hat{\mathbf{e}} \leftarrow \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\hat{m}}$ in the challenge ciphertext: in $\mathsf{Hyb}_2^{(b)}$, the challenger resets $\hat{\mathbf{e}} = \mathbf{0}^{\hat{m}}$ whenever $\|\hat{\mathbf{e}}\| \geq \sqrt{\lambda}\,\widetilde{\sigma}$, whereas in $\mathsf{Hyb}_3^{(b)}$, the challenger does not check the norm of $\hat{\mathbf{e}}$. In particular, the two experiments proceed identically unless $\|\hat{\mathbf{e}}\| \geq \sqrt{\lambda}\,\widetilde{\sigma}$. Each entry of $\hat{\mathbf{e}}$ is an independent sample from $\mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}$, so by the Gaussian tail bound (Lemma 3.2) and a union bound over the $\hat{m}$ entries of $\hat{\mathbf{e}}$,

$$\Pr\left[\|\hat{\mathbf{e}}\| \geq \sqrt{\lambda}\,\widetilde{\sigma}\right] \leq \hat{m} \cdot 2^{-\lambda}.$$

Since $\hat{m} = 4m^5 = \mathsf{poly}(\lambda)$, the statistical distance between $\mathsf{Hyb}_2^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_3^{(b)}(\mathcal{A})$ is at most $\hat{m} \cdot 2^{-\lambda} = \mathsf{negl}(\lambda)$. □

**Claim 4.19.** Suppose the matrix commitment scheme (Fact 3.5) is secure. Then, for all efficient adversaries $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_3^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_4^{(b)}(\mathcal{A})$ are computationally indistinguishable.

*Proof.* The only difference between the two experiments is the construction of the challenge ciphertext: in $\mathsf{Hyb}_3^{(b)}$, the challenger samples $\hat{\mathbf{s}} \xleftarrow{R} \mathbb{Z}_q^n$ and $\hat{\mathbf{e}} \leftarrow \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\hat{m}}$ and sets

$$\hat{\mathbf{c}}_1^\intercal = \hat{\mathbf{s}}^\intercal\mathbf{B} + \hat{\mathbf{e}}^\intercal, \qquad \hat{\mathbf{c}}_2^\intercal = \hat{\mathbf{s}}^\intercal(\mathbf{A} + \hat{\mathbf{C}}) + \hat{\mathbf{e}}^\intercal\hat{\mathbf{D}}_1, \qquad \hat{c}_3 = \hat{\mathbf{s}}^\intercal\mathbf{p} + \hat{\mathbf{e}}^\intercal\hat{\mathbf{d}}_2 + \xi \cdot b,$$

whereas in $\mathsf{Hyb}_4^{(b)}$, the challenger samples $\hat{\mathbf{c}}_1 \xleftarrow{R} \mathbb{Z}_q^{\hat{m}}$ and sets $\hat{\mathbf{c}}_2^\intercal = \hat{\mathbf{c}}_1^\intercal\hat{\mathbf{D}}_1$ and $\hat{c}_3 = \hat{\mathbf{c}}_1^\intercal\hat{\mathbf{d}}_2 + \xi \cdot b$. Fix a bit $b \in \{0,1\}$. Suppose there exists an efficient adversary $\mathcal{A}$ where

$$\left|\Pr[\mathsf{Hyb}_3^{(b)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_4^{(b)}(\mathcal{A}) = 1]\right| \geq \varepsilon(\lambda),$$

for some non-negligible $\varepsilon$. We use $\mathcal{A}$ to construct an adversary $\mathcal{B}$ for the matrix-commitment security game (Fact 3.5):

- **Initialization:** At the beginning of the game, algorithm $\mathcal{B}$ receives a challenge $(\mathsf{pp}_{\mathsf{com}}, \hat{\mathbf{y}})$ from the matrix-commitment challenger, where $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$ and $\hat{\mathbf{y}} \in \mathbb{Z}_q^{\hat{m}}$ is either an LWE sample $\hat{\mathbf{y}}^\intercal = \hat{\mathbf{s}}^\intercal\mathbf{B} + \hat{\mathbf{e}}^\intercal$ (where $\hat{\mathbf{s}} \xleftarrow{R} \mathbb{Z}_q^n$ and $\hat{\mathbf{e}} \leftarrow \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\hat{m}}$) or a uniform random vector $\hat{\mathbf{y}} \xleftarrow{R} \mathbb{Z}_q^{\hat{m}}$.
- **Static commitment phase:** Algorithm $\mathcal{B}$ starts running algorithm $\mathcal{A}$, which commits to the entire sequence of pre-challenge queries it will make.
- **Setup phase:** Algorithm $\mathcal{B}$ computes the sets $\hat{T}$ and $C$ determined by the committed sequence of queries as in $\mathsf{Hyb}_3^{(b)}$ and $\mathsf{Hyb}_4^{(b)}$. If $\hat{T} = \varnothing$ or $\hat{T} \cap C \neq \varnothing$, algorithm $\mathcal{B}$ halts with output 0. Algorithm $\mathcal{B}$ uses the parameters $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$ from its challenge instead of sampling them itself. It then samples the public keys $\mathbf{t}_c \xleftarrow{R} \mathbb{Z}_q^n$ for all $c \in \hat{T}$ together with $\hat{\mathbf{D}}_1 \xleftarrow{R} \{0,1\}^{\hat{m}\times m}$ and $\hat{\mathbf{d}}_2 \xleftarrow{R} \{0,1\}^{\hat{m}}$ and computes

  $$\hat{\mathbf{C}} = \sum_{c\in\hat{T}} \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_c}^\intercal \otimes \mathbf{t}_c),$$

  where $i_c$ denotes the index associated with counter $c$ in the committed sequence of queries. Algorithm $\mathcal{B}$ then sets $\mathbf{A} = \mathbf{B}\hat{\mathbf{D}}_1 - \hat{\mathbf{C}}$ and $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_2$. Next, algorithm $\mathcal{B}$ initializes a counter $\mathsf{ctr} = 0$, a dictionary $\mathsf{D} = \varnothing$, a mapping $S = \varnothing$, a set $T = \varnothing$, the aggregated public key $\widetilde{\mathbf{C}} = \mathbf{0}^{n\times m}$, and a set of corrupted counters $C = \varnothing$, and processes the committed queries in order as follows:

– **Key-generation query on an index $i \in [2^\lambda]$:** Algorithm $\mathcal{B}$ increments the counter ctr = ctr+1. If ctr $\in \hat{T}$, it sets $\mathsf{pk}_{\mathsf{ctr}} = \mathbf{t}_{\mathsf{ctr}}$ (the public key sampled at the beginning of the setup phase) and $\mathsf{sk}_{\mathsf{ctr}} = \mathsf{ask}_{\mathsf{ctr}} = \perp$. If ctr $\notin \hat{T}$, it samples $\mathbf{r}_{\mathsf{ctr}} \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$, computes

$$\mathbf{v}_i = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i), \qquad \mathbf{t}_{\mathsf{ctr}} = \mathbf{B}\mathbf{r}_{\mathsf{ctr}} + \mathbf{p} - \mathbf{A}\mathbf{v}_i, \qquad \mathbf{z}_{\mathsf{ctr}} = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes \mathbf{t}_{\mathsf{ctr}}, i),$$

and sets $\mathsf{pk}_{\mathsf{ctr}} = \mathbf{t}_{\mathsf{ctr}}$, $\mathsf{sk}_{\mathsf{ctr}} = (i, \mathbf{r}_{\mathsf{ctr}}, \mathbf{z}_{\mathsf{ctr}})$, and $\mathsf{ask}_{\mathsf{ctr}} = \perp$. In both cases, algorithm $\mathcal{B}$ adds the mapping $\mathsf{ctr} \mapsto (i, \mathsf{pk}_{\mathsf{ctr}}, \mathsf{sk}_{\mathsf{ctr}}, \mathsf{ask}_{\mathsf{ctr}})$ to D and defines the response to the query to be $\mathsf{pk}_{\mathsf{ctr}}$.

– **Corruption query on a counter $c \in [\mathsf{ctr}]$:** Algorithm $\mathcal{B}$ looks up $(i, \mathsf{pk}_c, \mathsf{sk}_c, \mathsf{ask}_c) = \mathsf{D}[c]$, defines the response to the query to be $(\mathsf{sk}_c, \mathsf{ask}_c)$, and updates $C = C \cup \{c\}$.

– **Update query on a counter $c \in [\mathsf{ctr}]$ and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$:** Algorithm $\mathcal{B}$ looks up $(i, \mathsf{pk}_c, \mathsf{sk}_c, \mathsf{ask}_c) = \mathsf{D}[c]$ and writes $\mathsf{pk}_c = \mathbf{t}_c$. If either (op = Add and $i \in S$) or (op = Remove and $c \notin T$), algorithm $\mathcal{B}$ defines the response to this query to be $\perp$. Otherwise, it computes the commitment

$$\mathbf{C}_{(c)} = \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes \mathbf{t}_c) \in \mathbb{Z}_q^{n \times m}.$$

As in $\mathsf{Hyb}_3^{(b)}$ and $\mathsf{Hyb}_4^{(b)}$, for counters $\tau, \tau' \in T \cup \{c\}$, we write $(i_\tau, \mathbf{t}_\tau)$ to denote the index and public key associated with $\mathsf{D}[\tau]$, and we write

$$\mathbf{z}_{(\tau,\tau')} = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_\tau}^\intercal \otimes \mathbf{t}_\tau, i_{\tau'})$$

to denote their respective local openings. Note that the algorithms Com and Open are deterministic, and algorithm $\mathcal{B}$ knows all of their inputs (the public keys stored in D). Algorithm $\mathcal{B}$ then proceeds as follows based on the operation:

* **Case** op = Add**:** For each $\tau \in T \setminus \hat{T}$, algorithm $\mathcal{B}$ looks up $(i_\tau, \mathsf{pk}_\tau, \mathsf{sk}_\tau, \mathsf{ask}_\tau) = \mathsf{D}[\tau]$, parses $\mathsf{ask}_\tau = (i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau)$, and updates $\mathsf{ask}_\tau$ in $\mathsf{D}[\tau]$ with $(i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau + \mathbf{z}_{(c,\tau)})$. If $c \notin \hat{T}$, algorithm $\mathcal{B}$ parses $\mathsf{sk}_c = (i, \mathbf{r}_c, \mathbf{z}_c)$, computes

$$\mathsf{ask}_c = \Big(i, \mathbf{r}_c, \mathbf{z}_c + \sum_{\tau \in T} \mathbf{z}_{(\tau,c)}\Big),$$

and updates $\mathsf{ask}_c$ in $\mathsf{D}[c]$. Finally, algorithm $\mathcal{B}$ updates the aggregated public key $\widetilde{\mathbf{C}} = \widetilde{\mathbf{C}} + \mathbf{C}_{(c)}$, sets $S[i] = \mathbf{t}_c$, and updates $T = T \cup \{c\}$.

* **Case** op = Remove**:** For each $\tau \in T \setminus \{c\}$ where $\tau \notin \hat{T}$, algorithm $\mathcal{B}$ looks up $(i_\tau, \mathsf{pk}_\tau, \mathsf{sk}_\tau, \mathsf{ask}_\tau) = \mathsf{D}[\tau]$, parses $\mathsf{ask}_\tau = (i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau)$, and updates $\mathsf{ask}_\tau = (i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau - \mathbf{z}_{(c,\tau)})$ in $\mathsf{D}[\tau]$. It also sets $\mathsf{ask}_c = \perp$ in $\mathsf{D}[c]$. Finally, algorithm $\mathcal{B}$ updates the aggregated public key $\widetilde{\mathbf{C}} = \widetilde{\mathbf{C}} - \mathbf{C}_{(c)}$, removes $i$ from $S$, and updates $T = T \setminus \{c\}$.

As in $\mathsf{Hyb}_3^{(b)}$ and $\mathsf{Hyb}_4^{(b)}$, algorithm $\mathcal{B}$ does not update the aggregated secret keys for the counters in $\hat{T}$; these remain $\perp$. Algorithm $\mathcal{B}$ defines the response to the query to be $(\widetilde{\mathbf{C}}, \{(\tau, \mathsf{ask}_\tau)\}_{\tau \in T \cap C})$.

- **Challenge ciphertext:** Algorithm $\mathcal{B}$ constructs the challenge ciphertext $\mathsf{ct}^* = (\hat{\mathbf{c}}_1, \hat{\mathbf{c}}_2, \hat{c}_3)$ where

$$\hat{\mathbf{c}}_1 = \hat{\mathbf{y}}, \qquad \hat{\mathbf{c}}_2^\intercal = \hat{\mathbf{y}}^\intercal \hat{\mathbf{D}}_1, \qquad \hat{c}_3 = \hat{\mathbf{y}}^\intercal \hat{\mathbf{d}}_2 + \xi \cdot b.$$

It gives the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, the responses to the committed pre-challenge queries, and the challenge ciphertext $\mathsf{ct}^*$ to $\mathcal{A}$.

- **Post-challenge query phase:** Algorithm $\mathcal{B}$ responds to each of the adversary's adaptive key-generation, corruption, and update queries using the same procedure it used to process the committed pre-challenge queries.

- **Output phase:** At the end of the experiment, algorithm $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$, which algorithm $\mathcal{B}$ also outputs.

Since $\mathcal{A}$ is efficient, algorithm $\mathcal{B}$ is also efficient. By construction, algorithm $\mathcal{B}$ implements the challenger in $\mathsf{Hyb}_3^{(b)}$ and $\mathsf{Hyb}_4^{(b)}$ exactly, except it uses its challenge $\hat{\mathbf{y}}$ to construct the challenge ciphertext. Other than the construction of the challenge ciphertext, the challenger's behavior in the two experiments is identical and does not depend on $\hat{\mathbf{s}}$ or $\hat{\mathbf{e}}$. We now consider the two possibilities for $\hat{\mathbf{y}}$:

- Suppose $\hat{\mathbf{y}}^\intercal = \hat{\mathbf{s}}^\intercal \mathbf{B} + \hat{\mathbf{e}}^\intercal$ where $\hat{\mathbf{s}} \xleftarrow{R} \mathbb{Z}_q^n$ and $\hat{\mathbf{e}} \leftarrow \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\hat{m}}$. Since $\mathbf{A} + \hat{\mathbf{C}} = \mathbf{B}\hat{\mathbf{D}}_1$ and $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_2$, this means

$$\begin{aligned}
\hat{\mathbf{c}}_1^\intercal &= \hat{\mathbf{y}}^\intercal = \hat{\mathbf{s}}^\intercal \mathbf{B} + \hat{\mathbf{e}}^\intercal \\
\hat{\mathbf{c}}_2^\intercal &= \hat{\mathbf{y}}^\intercal \hat{\mathbf{D}}_1 = \left(\hat{\mathbf{s}}^\intercal \mathbf{B} + \hat{\mathbf{e}}^\intercal\right)\hat{\mathbf{D}}_1 = \hat{\mathbf{s}}^\intercal(\mathbf{A} + \hat{\mathbf{C}}) + \hat{\mathbf{e}}^\intercal \hat{\mathbf{D}}_1 \\
\hat{c}_3 &= \hat{\mathbf{s}}^\intercal \mathbf{p} + \hat{\mathbf{e}}^\intercal \hat{\mathbf{d}}_2 + \xi \cdot b = \left(\hat{\mathbf{s}}^\intercal \mathbf{B} + \hat{\mathbf{e}}^\intercal\right)\hat{\mathbf{d}}_2 + \xi \cdot b = \hat{\mathbf{y}}^\intercal \hat{\mathbf{d}}_2 + \xi \cdot b.
\end{aligned}$$

  Thus, in this case, algorithm $\mathcal{B}$ constructs the challenge ciphertext exactly according to the specification in $\mathsf{Hyb}_3^{(b)}$.

- Suppose $\hat{\mathbf{y}} \xleftarrow{R} \mathbb{Z}_q^{\hat{m}}$. Then $\hat{\mathbf{c}}_1 = \hat{\mathbf{y}}$ is uniform over $\mathbb{Z}_q^{\hat{m}}$, and algorithm $\mathcal{B}$ sets $\hat{\mathbf{c}}_2^\intercal = \hat{\mathbf{c}}_1^\intercal \hat{\mathbf{D}}_1$ and $\hat{c}_3 = \hat{\mathbf{c}}_1^\intercal \hat{\mathbf{d}}_2 + \xi \cdot b$. This matches the distribution in $\mathsf{Hyb}_4^{(b)}$.

We conclude that the distinguishing advantage of $\mathcal{B}$ in the matrix-commitment security game is at least $\varepsilon$, which contradicts the security of the matrix commitment scheme (Fact 3.5). □

**Claim 4.20.** Suppose $n \geq \lambda$, $m \geq 2(n+1)\log q$, $\sigma \geq \log m$, and $q$ is prime. Then, for all adversaries $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_4^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_5^{(b)}(\mathcal{A})$ are statistically indistinguishable.

*Proof.* The only difference between the two experiments is the distribution of the pair $(\mathbf{p}, \hat{c}_3)$:

- In $\mathsf{Hyb}_4^{(b)}$, the challenger sets $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_2$ and $\hat{c}_3 = \hat{\mathbf{c}}_1^\intercal \hat{\mathbf{d}}_2 + \xi \cdot b$, where $\hat{\mathbf{d}}_2 \xleftarrow{R} \{0,1\}^{\hat{m}}$.
- In $\mathsf{Hyb}_5^{(b)}$, the challenger samples $\mathbf{p} \xleftarrow{R} \mathbb{Z}_q^n$ and $\hat{c}_3 \xleftarrow{R} \mathbb{Z}_q$.

Next, no other quantity in $\mathsf{Hyb}_4^{(b)}$ depends on the vector $\hat{\mathbf{d}}_2$. The claim then follows by two invocations of the leftover hash lemma:

- In $\mathsf{Hyb}_4^{(b)}$, the challenger samples $\hat{\mathbf{c}}_1 \xleftarrow{R} \mathbb{Z}_q^{\hat{m}}$ (which is independent of all other parameters). By Lemma 3.4, the distributions

$$\left(\mathbf{W},\ \mathbf{R},\ \hat{\mathbf{c}}_1,\ \mathbf{B}\hat{\mathbf{d}}_2,\ \hat{\mathbf{c}}_1^\intercal \hat{\mathbf{d}}_2\right) \qquad \text{and} \qquad \left(\mathbf{W},\ \mathbf{R},\ \hat{\mathbf{c}}_1,\ \mathbf{p},\ \hat{\mathbf{c}}_1^\intercal \hat{\mathbf{d}}_2\right)$$

  are statistically close, where $\mathbf{p} \xleftarrow{R} \mathbb{Z}_q^n$.

- Next, by Lemma 3.1, the distributions $(\hat{\mathbf{c}}_1^\intercal, \hat{\mathbf{c}}_1^\intercal \hat{\mathbf{d}}_2)$ and $(\hat{\mathbf{c}}_1^\intercal, \hat{c}_3)$ are statistically close, where $\hat{c}_3 \xleftarrow{R} \mathbb{Z}_q$.

By a standard hybrid argument, this means the following distributions are statistically indistinguishable:

$$\left(\mathbf{W},\ \mathbf{R},\ \hat{\mathbf{c}}_1,\ \mathbf{B}\hat{\mathbf{d}}_2,\ \hat{\mathbf{c}}_1^\intercal \hat{\mathbf{d}}_2 + \xi \cdot b\right) \qquad \text{and} \qquad \left(\mathbf{W},\ \mathbf{R},\ \hat{\mathbf{c}}_1,\ \mathbf{p},\ \hat{c}_3\right).$$

The left distribution corresponds to $\mathsf{Hyb}_4^{(b)}$ while the right corresponds to $\mathsf{Hyb}_5^{(b)}$. □

**Completing the proof.** By definition, the challenger's behavior in $\mathsf{Hyb}_5^{(b)}$ is independent of the bit $b \in \{0,1\}$. Thus, $\mathsf{Hyb}_5^{(0)}$ and $\mathsf{Hyb}_5^{(1)}$ are identical experiments. Combining Claims 4.16 to 4.20, static security now follows by a standard hybrid argument. □

**Parameter instantiation.** We conclude by describing one way to instantiate the parameters of Construction 4.11 to satisfy the requirements of Theorems 4.12, 4.14 and 4.15. Take any constant $0 < \varepsilon < 1$ and choose the lattice parameters $(n, m, q, \widetilde{\sigma})$ and the width parameter $\sigma$ such that

$$n = \lambda^{1/\varepsilon} \;;\; q = 2^{\Theta(\lambda)} \;;\; m = 2(n+1)\lceil \log q \rceil \;;\; \widetilde{\sigma} = \mathsf{poly}(\lambda) \;;\; \sigma = \mathsf{poly}(\lambda),$$

where $q$ is prime, the constant in $q = 2^{\Theta(\lambda)}$ is chosen large enough so that $q/\log q > 2^{\lambda} \cdot \mathsf{poly}(\lambda)$ (for instance, the constant 2 is enough), and the $(2m^2, \sigma)$-decomposed LWE assumption holds with parameters $(n, m, q, \widetilde{\sigma})$. We now affirm that this choice of parameters satisfies our requirements:

- **Correctness:** $n, m, \sigma, \widetilde{\sigma} = \mathsf{poly}(\lambda)$ and $q/\log q > 2^{\lambda} \cdot \mathsf{poly}(\lambda)$, so Theorem 4.12 is satisfied.
- **Succinctness:** $n, m = \mathsf{poly}(\lambda)$ and $q \leq 2^{\mathsf{poly}(\lambda)}$, so Theorem 4.14 is satisfied.
- **Security:** $n \geq \lambda$, $m \geq 2(n+1)\log q$, $\sigma \geq \log m$, $q$ is prime, and the $(2m^2, \sigma)$-decomposed LWE assumption with parameters $(n, m, q, \widetilde{\sigma})$ holds. Then, the matrix commitment scheme from Fact 3.5 is secure and Theorem 4.15 is satisfied.

This instantiation relies on the polynomial hardness of the decomposed LWE assumption with a sub-exponential modulus-to-noise ratio $q/\widetilde{\sigma} = 2^{\Theta(\lambda)} = 2^{O(n^{\varepsilon})}$. We summarize this instantiation with the following corollary:

**Corollary 4.21** (Incremental DBE from Decomposed LWE)**.** Assuming the polynomial hardness of the decomposed LWE assumption (Assumption 3.3) with a sub-exponential modulus-to-noise ratio, Construction 4.11 is a statically secure incremental distributed broadcast encryption scheme.

# 5 Updatable Distributed Broadcast Encryption

Our notion of *updatable* distributed broadcast encryption extends incremental distributed broadcast encryption by additionally providing *forward secrecy*. An updatable DBE scheme consists of a tuple of efficient algorithms $\Pi_{\mathsf{UpdDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{UpdPK}, \mathsf{UpdSK})$. The key distinction from an incremental DBE scheme is a key-rotation mechanism using the algorithms UpdPK and UpdSK. We begin with a brief overview of how these algorithms map to the group messaging setting.

- **Setup, key generation, membership updates, encryption, and decryption.** These algorithms are identical in structure to those of an incremental distributed broadcast encryption scheme. Each group member maintains a copy of the aggregated public encryption key apk and their individual aggregated secret key $\mathsf{ask}_i$. As in an incremental distributed broadcast encryption scheme, group members update the public key and their aggregated secret key using IncPK and IncSK when users join or leave the group. As before, to send a message $m$ to the current set of users, the sender runs the encryption algorithm Enc using the current aggregated public key apk and produces a ciphertext ct. Upon receiving ct, each recipient $i$ runs the decryption algorithm Dec with their aggregated secret key $\mathsf{ask}_i$.
- **Key rotation and forward secrecy.** An important security requirement in secure group messaging is forward secrecy. Namely, a future compromise of one or more users should not jeopardize confidentiality of past messages. To ensure this, we introduce an explicit key rotation system, where senders also include an encrypted "update" message whenever they send a message to the group. Each group member in turn refreshes their public and secret keys using the encrypted update.

  More formally, to prepare an encrypted update message, the sender (in the group) runs the UpdPK algorithm with the current aggregated public key and the current set of group members. This algorithm outputs an encrypted update ctref, a new group public key apk′, and the updated public keys associated with each group member. Upon receiving ctref, each recipient $i$ runs the UpdSK algorithm with their current aggregated secret key $\mathsf{ask}_i$, their public and secret keys $\mathsf{pk}_i, \mathsf{sk}_i$, and the current set of member public keys. The UpdSK algorithm outputs a refreshed aggregated secret key $\mathsf{ask}'_i$ and secret key $\mathsf{sk}'_i$, together with the updated set of member

public keys (which contains the user's updated individual public key $\mathsf{pk}_i'$, consistent with the updated group public key $\mathsf{apk}'$). In the group messaging application, after updating their secret key from $\mathsf{ask}_i$ to $\mathsf{ask}_i'$, each decrypter then erases their previous secret key $\mathsf{ask}_i$. If a refresh ciphertext is sent along with every normal ciphertext, then each user only ever uses their aggregated decryption key to decrypt a single message. This ensures forward secrecy. Formally, we capture this by defining an "updatable" security property on the updatable distributed broadcast encryption scheme.

We now give the formal syntax and security requirements.

**Definition 5.1** (Updatable Distributed Broadcast Encryption)**.** An updatable distributed broadcast encryption scheme is a tuple of efficient algorithms $\Pi_{\mathsf{UpdDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{UpdPK}, \mathsf{UpdSK})$ with the following syntax:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$: On input the security parameter $\lambda$, the setup algorithm outputs a set of public parameters pp.
- $\mathsf{KeyGen}(\mathsf{pp}, i) \to (\mathsf{pk}_i, \mathsf{sk}_i)$: On input the public parameters pp and an index $i \in [2^\lambda]$, the key-generation algorithm outputs a public key $\mathsf{pk}_i$ and a secret key $\mathsf{sk}_i$.
- $\mathsf{IncPK}(\mathsf{pp}, \mathsf{apk}, (i, \mathsf{pk}_i), \mathsf{op}) \to \mathsf{apk}'$: On input the public parameters pp, an aggregated public key apk, a public key $\mathsf{pk}_i$ (with index $i$), and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, the algorithm computes a new aggregated public key. This algorithm is deterministic.
- $\mathsf{IncSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, (i, \mathsf{pk}_i), \mathsf{op}) \to \mathsf{ask}_{i^*}'$: On input the public parameters pp, an aggregated secret key for the current set, a new public key, and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, the algorithm computes the new aggregated secret key. This algorithm is deterministic.
- $\mathsf{Enc}(\mathsf{pp}, \mathsf{apk}, \mu) \to \mathsf{ct}$: On input the public parameters pp, an aggregated public key apk and a message $\mu \in \{0, 1\}$, the encryption algorithm outputs a ciphertext ct.
- $\mathsf{Dec}(\mathsf{pp}, \mathsf{ask}_i, \mathsf{ct}) \to \mu$: On input the public parameters pp, an aggregated secret key $\mathsf{ask}_i$ and a ciphertext ct, the decryption algorithm outputs a message $\mu \in \{0, 1\}$.
- $\mathsf{UpdPK}(\mathsf{pp}, \mathsf{apk}, S) \to (\mathsf{ctref}, \mathsf{apk}', S')$: On input the public parameters pp, an aggregated public key apk, the current set of group members $S$ represented as a mapping from the index $j$ of a group member to their public key $\mathsf{pk}_j$, this algorithm outputs a refresh ciphertext ctref, an updated aggregated public key $\mathsf{apk}'$, and an updated mapping $S'$ containing the refreshed public keys of the group members.
- $\mathsf{UpdSK}(\mathsf{pp}, \mathsf{ask}_i, \mathsf{ctref}, S, \mathsf{pk}_i, \mathsf{sk}_i) \to (\mathsf{ask}_i', \mathsf{pk}_i', \mathsf{sk}_i', S')$: On input the public parameters pp, an aggregated secret key $\mathsf{ask}_i$, a refresh ciphertext ctref, the current set of group members $S$ represented as a mapping from the index $j$ of a group member to their public key $\mathsf{pk}_j$, and public, secret keys $\mathsf{pk}_i, \mathsf{sk}_i$ for user $i$, this algorithm outputs an updated aggregated secret key $\mathsf{ask}_i'$, updated public, secret keys $\mathsf{pk}_i', \mathsf{sk}_i'$, and an updated mapping $S'$.

**Correctness.** The correctness requirement for an updatable DBE scheme states that for any sequence of add, remove, and *refresh* operations, whenever a user encrypts a message to the current group, all users currently in the group can decrypt it using their respective aggregated secret keys. To formally define correctness (similar to Section 4), we define two helper subroutines: (1) an UpdateAPK subroutine that tracks the current aggregated public key associated with the group; and (2) an UpdateASK algorithm that tracks the aggregated secret key associated with a particular user. We use these subroutines to streamline our correctness and security definitions. The routines are the analogs of those defined in Definitions 4.2 and 4.3, except extended to support the additional refresh operation. We define the syntax formally below:

**Definition 5.2** (UpdateAPK Subroutine for Updatable DBE)**.** Let $\Pi_{\mathsf{UpdDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{UpdPK}, \mathsf{UpdSK})$ be an updatable distributed broadcast encryption scheme. We define the UpdateAPK subroutine as follows:

$\mathsf{UpdateAPK}(\mathsf{pp}, \mathsf{apk}, (i, \mathsf{pk}_i), \mathsf{op}, S)$

**Inputs:** Public parameters pp, an aggregated public key apk, index $i$, a public key $\mathsf{pk}_i$, an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{Refresh}\}$, and a mapping $S$ that maps an index $j$ (of a group member) to a public key $\mathsf{pk}_j$.

**Outputs:** Updated aggregated public key apk, updated mapping $S$, and in the case of a refresh operation, a pair $(\mathsf{ctref}, \rho)$ consisting of a refresh ciphertext and the randomness $\rho$ used to generate it.

**Computation:**

- If $\mathsf{op} = \mathsf{Add}$ and $i \notin S$, compute $\mathsf{apk} = \mathsf{IncPK}(\mathsf{pp}, \mathsf{apk}, (i, \mathsf{pk}_i), \mathsf{Add})$. Set $S[i] = \mathsf{pk}_i$ and return $(\mathsf{apk}, S, \perp)$.
- If $\mathsf{op} = \mathsf{Remove}$ and $i \in S$, compute $\mathsf{apk} = \mathsf{IncPK}(\mathsf{pp}, \mathsf{apk}, (i, S[i]), \mathsf{Remove})$. Remove $i$ from $S$ and return $(\mathsf{apk}, S, \perp)$.
- If $\mathsf{op} = \mathsf{Refresh}$, then compute $(\mathsf{ctref}, \mathsf{apk}, S) \leftarrow \mathsf{UpdPK}(\mathsf{pp}, \mathsf{apk}, S; \rho)$ where $\rho$ is the random coins used by the algorithm. Return $(\mathsf{apk}, S, (\mathsf{ctref}, \rho))$.
- In all other cases, return $(\mathsf{apk}, S, \perp)$.

Next, we define the corresponding UpdateASK subroutine that keeps track of a user's aggregate secret key as users join, leave, or are refreshed in the group. The UpdateASK subroutine takes the following inputs:

- the public parameters pp,
- the current set of group members $S$, represented as a mapping from the index $j$ of a group member to their public key $\mathsf{pk}_j$,
- the public key $\mathsf{pk}_{i^*}$, the secret key $\mathsf{sk}_{i^*}$, and the aggregated secret key $\mathsf{ask}_{i^*}$ for the user performing the update (i.e., user $i^*$),
- the public key $\mathsf{pk}_i$ of the user joining/leaving the group,
- auxiliary refresh information consisting of a refresh ciphertext ctref, and
- the description of the operation op (i.e., specifying whether user $i$ is joining the group, leaving the group, or refreshing the group public keys).

The UpdateASK subroutine then outputs the updated aggregated secret key $\mathsf{ask}_{i^*}$, the updated mapping $S_{i^*}$ maintained by user $i^*$, the updated public key $\mathsf{pk}_{i^*}$, and the updated secret key $\mathsf{sk}_{i^*}$ associated with user $i^*$. In the correctness and security definitions, the challenger uses UpdateASK to track the state of each user's keys. We now describe the formal syntax and behavior and then give our formal correctness definition:

**Definition 5.3** (UpdateASK Subroutine for Updatable DBE)**.** Let $\Pi_{\mathsf{UpdDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{UpdPK}, \mathsf{UpdSK})$ be an updatable distributed broadcast encryption scheme. We define the UpdateASK subroutine as follows:

$\mathsf{UpdateASK}(\mathsf{pp}, (i^*, \mathsf{pk}_{i^*}, \mathsf{sk}_{i^*}, \mathsf{ask}_{i^*}), (i, \mathsf{pk}_i), \mathsf{ctref}, \mathsf{op}, S)$

**Inputs:** Public parameters pp, a public key $\mathsf{pk}_{i^*}$, secret key $\mathsf{sk}_{i^*}$, and aggregated secret key $\mathsf{ask}_{i^*}$ for user $i^*$, a public key $\mathsf{pk}_i$ for user $i$, auxiliary refresh information ctref, an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{Refresh}\}$, and a mapping $S$ that maps an index $j$ (of a group member) to a public key $\mathsf{pk}_j$.

**Outputs:** Updated aggregated secret key $\mathsf{ask}_{i^*}$, updated local mapping $S_{i^*}$, updated public key $\mathsf{pk}_{i^*}$, and updated secret key $\mathsf{sk}_{i^*}$.

**Computation:** First, initialize $S_{i^*} = S$. Then if $\mathsf{op} = \mathsf{Add}$ or $\mathsf{op} = \mathsf{Remove}$, perform the following updates:

- If $\mathsf{op} = \mathsf{Add}$ and $i \notin S$, set $S_{i^*}[i] = \mathsf{pk}_i$.
- If $\mathsf{op} = \mathsf{Remove}$ and $i \in S$, remove $i$ from $S_{i^*}$.

Next, perform the following updates:

- If $\mathsf{op} = \mathsf{Add}$, $i \notin S$, and $i^* = i$, initialize $\mathsf{ask}_{i^*} = \mathsf{sk}_{i^*}$, and for each $j \in S$ in lexicographic order, run $\mathsf{ask}_{i^*} = \mathsf{IncSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, (j, S[j]), \mathsf{Add})$.
- If $\mathsf{op} = \mathsf{Add}$, $i \notin S$, and $i^* \in S$, set $\mathsf{ask}_{i^*} = \mathsf{IncSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, (i, \mathsf{pk}_i), \mathsf{Add})$.
- If $\mathsf{op} = \mathsf{Remove}$, $i \in S$, and $i^* = i$, set $\mathsf{ask}_{i^*} = \perp$.
- If $\mathsf{op} = \mathsf{Remove}$, $i \in S$, $i^* \in S$, and $i^* \neq i$, set $\mathsf{ask}_{i^*} = \mathsf{IncSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, (i, S[i]), \mathsf{Remove})$.
- If $\mathsf{op} = \mathsf{Refresh}$ and $i^* \in S$, compute $(\mathsf{ask}_{i^*}, \mathsf{pk}_{i^*}, \mathsf{sk}_{i^*}, S_{i^*}) \leftarrow \mathsf{UpdSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, \mathsf{ctref}, S, \mathsf{pk}_{i^*}, \mathsf{sk}_{i^*})$.
- In all other cases, $\mathsf{ask}_{i^*}$, $\mathsf{pk}_{i^*}$, $\mathsf{sk}_{i^*}$ are unchanged.

Return $(\mathsf{ask}_{i^*}, S_{i^*}, \mathsf{pk}_{i^*}, \mathsf{sk}_{i^*})$.

**Definition 5.4** (Correctness)**.** Let $\Pi_{\mathsf{UpdDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{UpdPK}, \mathsf{UpdSK})$ be an updatable distributed broadcast encryption scheme. Let $\lambda \in \mathbb{N}$ be a security parameter, $Q \in \mathbb{N}$ be the number of queries, pp be a set of public parameters, and $\{(\mathsf{ind}_j, \mathsf{pk}_j, \mathsf{op}_j)\}_{j \in [Q]}$ be a sequence of queries where $\mathsf{op}_j \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{Refresh}\}$ and where the public key $\mathsf{pk}_j$ is used only when $\mathsf{op}_j = \mathsf{Add}$; we set $\mathsf{pk}_j = \perp$ whenever $\mathsf{op}_j \in \{\mathsf{Remove}, \mathsf{Refresh}\}$. For an index $i^* \in [2^\lambda]$ and a public/secret key-pair $(\mathsf{pk}_{i^*}, \mathsf{sk}_{i^*})$, we define the tuple $(\mathsf{apk}_Q, \mathsf{ask}_{i^*,Q}, \mathsf{pk}_{i^*,Q}, \mathsf{sk}_{i^*,Q}, S_Q)$ associated with this sequence as follows:

- Initialize $\mathsf{apk}_0 = \perp$, $\mathsf{ask}_{i^*,0} = \perp$, $\mathsf{pk}_{i^*,0} = \mathsf{pk}_{i^*}$, $\mathsf{sk}_{i^*,0} = \mathsf{sk}_{i^*}$, $S_0 = \emptyset$, and $S_{i^*,0} = \emptyset$.
- For each $j \in [Q]$, compute

$$(\mathsf{apk}_j, S_j, (\mathsf{ctref}_j, \rho_j)) = \mathsf{UpdateAPK}(\mathsf{pp}, \mathsf{apk}_{j-1}, (\mathsf{ind}_j, \mathsf{pk}_j), \mathsf{op}_j, S_{j-1})$$
$$(\mathsf{ask}_{i^*,j}, S_{i^*,j}, \mathsf{pk}_{i^*,j}, \mathsf{sk}_{i^*,j}) = \mathsf{UpdateASK}(\mathsf{pp}, (i^*, \mathsf{pk}_{i^*,j-1}, \mathsf{sk}_{i^*,j-1}, \mathsf{ask}_{i^*,j-1}), (\mathsf{ind}_j, \mathsf{pk}_j), \mathsf{ctref}_j, \mathsf{op}_j, S_{j-1}).$$

- Output $(\mathsf{apk}_Q, \mathsf{ask}_{i^*,Q}, \mathsf{pk}_{i^*,Q}, \mathsf{sk}_{i^*,Q}, S_Q)$.

We say $\Pi_{\mathsf{UpdDBE}}$ is correct if for all security parameters $\lambda \in \mathbb{N}$, all pp in the support of $\mathsf{Setup}(1^\lambda)$, all $Q \leq 2^\lambda$, all indices $i^* \in [2^\lambda]$, all $(\mathsf{pk}_{i^*}, \mathsf{sk}_{i^*})$ in the support of $\mathsf{KeyGen}(\mathsf{pp}, i^*)$, all sequences $\{(\mathsf{ind}_j, \mathsf{pk}_j, \mathsf{op}_j)\}_{j \in [Q]}$ where $S_Q[i^*] = \mathsf{pk}_{i^*,Q}$, and all messages $\mu \in \{0,1\}$, we have the following two properties:

- **Mapping consistency:** For all $j \in [Q]$ such that $i^* \in S_j$, we have $S_{i^*,j} = S_j$. That is, the mapping maintained by user $i^*$ through UpdateASK agrees with the mapping maintained by UpdateAPK.
- **Decryption correctness:** It holds that

$$\Pr[\mathsf{Dec}(\mathsf{pp}, \mathsf{ask}_{i^*,Q}, \mathsf{ct}) = \mu : \mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{apk}_Q, \mu)] = 1,$$

  where $(\mathsf{apk}_Q, \mathsf{ask}_{i^*,Q}, \mathsf{pk}_{i^*,Q}, \mathsf{sk}_{i^*,Q}, S_Q)$ is the tuple associated with the sequence $\{(\mathsf{ind}_j, \mathsf{pk}_j, \mathsf{op}_j)\}_{j \in [Q]}$.

**Remark 5.5** (Mapping Consistency and the Correctness Analysis)**.** In a refresh operation, the UpdPK algorithm updates the public keys of all the users in the set. On the other hand, when users receive an update, each user locally runs UpdSK to update their individual public (and secret) key. The mapping consistency property ensures that these two operations produce the same view.

**Succinctness.** Our succinctness requirement stipulates that the size of the aggregate public key apk, the aggregate secret key ask, and the individual public and secret keys output by the IncPK, IncSK, UpdPK, and UpdSK algorithms are short (i.e., have size that is independent of the size of the group).

**Definition 5.6** (Succinctness)**.** There exists a universal polynomial $\mathsf{poly}(\cdot)$ such that for all security parameters $\lambda \in \mathbb{N}$, all public parameters pp in the support of $\mathsf{Setup}(1^\lambda)$, the size of the aggregated public key apk output by $\mathsf{IncPK}(\mathsf{pp}, \cdot, \cdot, \cdot)$ and $\mathsf{UpdPK}(\mathsf{pp}, \cdot, \cdot)$, the size of the aggregated secret key ask output by $\mathsf{IncSK}(\mathsf{pp}, \cdot, \cdot, \cdot)$ and $\mathsf{UpdSK}(\mathsf{pp}, \cdot, \cdot, \cdot, \cdot, \cdot)$, the size of the public and secret keys pk, sk output by $\mathsf{UpdSK}(\mathsf{pp}, \cdot, \cdot, \cdot, \cdot, \cdot)$, and the size of each public key $\mathsf{pk}_j$ in the mapping $S'$ output by $\mathsf{UpdPK}(\mathsf{pp}, \cdot, \cdot)$ and $\mathsf{UpdSK}(\mathsf{pp}, \cdot, \cdot, \cdot, \cdot, \cdot)$ are bounded by $\mathsf{poly}(\lambda)$. We also require that the size of the refresh ciphertext output by $\mathsf{UpdPK}(\mathsf{pp}, \cdot, \cdot)$ be $\mathsf{poly}(\lambda)$.

**Remark 5.7** (Efficiency of Refresh Operations)**.** The refresh algorithms UpdPK and UpdSK take the mapping $S$ as an explicit input, and as such, their running times are allowed to scale with the size of the group $|S|$. The only efficiency requirement we impose (see Definition 5.6) is that the size of the refresh ciphertext output by UpdPK be independent of the size of the group. In the context of group messaging (see Section 6), this gives a scheme with forward secrecy where the refresh operation takes time that is polynomial in the size of the group, but where communication is always *sublinear* in the size of the group (in fact, independent of the size of the group). An important open problem is to obtain a scheme where the refresh operation requires *sublinear* running time. An updatable distributed broadcast encryption scheme with this property would yield a forward-secure CGKA protocol where computation and communication costs are all sublinear in the size of the group.

We stress that the mapping $S'$ output by UpdPK and UpdSK is *local* state and is computed by each party from ctref and its local copy of $S$. This set $S'$ is *not* sent over the network. Updates only require transmitting the ciphertext ctref, whose size is $\mathsf{poly}(\lambda)$ by Definition 5.6. This is what ensures sublinear *communication* in our group messaging protocol (Section 6).

**Updatable DBE security.** We now define semantic security for updatable distributed broadcast encryption. In the *adaptive* definition, an adversary may dynamically add, remove, or refresh users in the group. In the *static* variant, we require the adversary to commit to its entire sequence of operations at the start of the game. We guarantee *forward security*. This intuitively means that an attacker that joins the group does not learn anything about past encrypted messages. In both static and adaptive cases, we require semantic security for any ciphertext encrypted with respect to the group public key provided that the adversary has not corrupted any user in the group at the time of the challenge; moreover, this holds even if we reveal the entire state of the system to the attacker after a single refresh operation. We now give the formal definition.

**Definition 5.8** (Security)**.** Let $\Pi_{\mathsf{UpdDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{UpdPK}, \mathsf{UpdSK})$ be an updatable distributed broadcast encryption scheme. For a security parameter $\lambda \in \mathbb{N}$, an adversary $\mathcal{A}$, and a challenge bit $b \in \{0, 1\}$, we define the updatable distributed broadcast encryption security game as follows:

- **Setup phase:** The challenger starts by sampling $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$ and initializes a counter $\mathsf{ctr} = 0$ and an (initially empty) dictionary D to keep track of key-generation queries. It also initializes a mapping $S = \varnothing$ that maps the index of each member of the current broadcast set to their public key, a set $T = \varnothing$ to keep track of the counters associated with the current broadcast set, the initial aggregated public key $\mathsf{apk} = \perp$, and a set of corrupted counters $C = \varnothing$.
- **Pre-challenge query phase:** The adversary may issue the following queries:
  - **Key-generation queries:** On input an index $i \in [2^\lambda]$, the challenger increments the counter $\mathsf{ctr} = \mathsf{ctr} + 1$ and samples $(\mathsf{pk}_{\mathsf{ctr}}, \mathsf{sk}_{\mathsf{ctr}}) \leftarrow \mathsf{KeyGen}(\mathsf{pp}, i)$. It also initializes an aggregate decryption key $\mathsf{ask}_{\mathsf{ctr}} = \perp$ and adds the mapping $\mathsf{ctr} \mapsto (i, \mathsf{pk}_{\mathsf{ctr}}, \mathsf{sk}_{\mathsf{ctr}}, \mathsf{ask}_{\mathsf{ctr}})$ to D. The challenger gives $\mathsf{pk}_{\mathsf{ctr}}$ to the adversary $\mathcal{A}$.
  - **Corruption queries:** On input a counter value $c \in [\mathsf{ctr}]$, the challenger looks up $(i, \mathsf{pk}_c, \mathsf{sk}_c, \mathsf{ask}_c) = \mathsf{D}[c]$ and returns $(\mathsf{sk}_c, \mathsf{ask}_c)$. The challenger then updates $C = C \cup \{c\}$.
  - **Update queries:** On input an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{Refresh}\}$ and, if $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, a counter value $c \in [\mathsf{ctr}]$ (if $\mathsf{op} = \mathsf{Refresh}$, no counter is required), the challenger proceeds as follows. If

$\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, it looks up $(i, \mathsf{pk}_c, \mathsf{sk}_c, \mathsf{ask}_c) = \mathsf{D}[c]$, checks the following conditions, and responds with $\perp$ if either holds:

* **Associating a key with an existing index $i$:** $\mathsf{op} = \mathsf{Add}$ and $i \in S$;
* **Removing a public key that is not in the group:** $\mathsf{op} = \mathsf{Remove}$ and $c \notin T$.

Otherwise, the challenger proceeds as follows:

* The challenger first computes the updated aggregated public key as follows:
  - If $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, compute $(\mathsf{apk}', S', (\mathsf{ctref}, \rho)) = \mathsf{UpdateAPK}(\mathsf{pp}, \mathsf{apk}, (i, \mathsf{pk}_c), \mathsf{op}, S)$.
  - If $\mathsf{op} = \mathsf{Refresh}$, compute $(\mathsf{apk}', S', (\mathsf{ctref}, \rho)) = \mathsf{UpdateAPK}(\mathsf{pp}, \mathsf{apk}, (\perp, \perp), \mathsf{Refresh}, S)$.
* If $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, then for each $t \in T \cup \{c\}$, the challenger looks up $(i_t, \mathsf{pk}_t, \mathsf{sk}_t, \mathsf{ask}_t) = \mathsf{D}[t]$ and computes the updated aggregate decryption key

$$(\mathsf{ask}_t, S_t, \mathsf{pk}_t, \mathsf{sk}_t) = \mathsf{UpdateASK}\big(\mathsf{pp}, (i_t, \mathsf{pk}_t, \mathsf{sk}_t, \mathsf{ask}_t), (i, \mathsf{pk}_c), \mathsf{ctref}, \mathsf{op}, S\big),$$

  using the previous mapping $S$. The challenger checks that $S_t = S'$ for each $t \in T \cup \{c\}$ and updates $\mathsf{D}[t] = (i_t, \mathsf{pk}_t, \mathsf{sk}_t, \mathsf{ask}_t)$. If the check fails, it halts the experiment and outputs 0.
* If $\mathsf{op} = \mathsf{Refresh}$, then for each $t \in T$, the challenger looks up $(i_t, \mathsf{pk}_t, \mathsf{sk}_t, \mathsf{ask}_t) = \mathsf{D}[t]$ and computes the updated aggregate decryption key

$$(\mathsf{ask}_t, S_t, \mathsf{pk}_t, \mathsf{sk}_t) = \mathsf{UpdateASK}\big(\mathsf{pp}, (i_t, \mathsf{pk}_t, \mathsf{sk}_t, \mathsf{ask}_t), (\perp, \perp), \mathsf{ctref}, \mathsf{op}, S\big),$$

  using the previous mapping $S$. The challenger checks that $S_t = S'$ for each $t \in T$ and updates $\mathsf{D}[t] = (i_t, \mathsf{pk}_t, \mathsf{sk}_t, \mathsf{ask}_t)$. If the check fails, it halts the experiment and outputs 0.
* The challenger then updates $\mathsf{apk} = \mathsf{apk}'$ and $S = S'$.
* If $\mathsf{op} = \mathsf{Add}$, then the challenger updates $T = T \cup \{c\}$. If $\mathsf{op} = \mathsf{Remove}$, then it updates $T = T \setminus \{c\}$.

The challenger responds with the updated aggregate public key apk, auxiliary update information $(\mathsf{ctref}, \rho)$ as well as the updated aggregate decryption key, public/secret key pair $\mathsf{ask}_t, \mathsf{pk}_t, \mathsf{sk}_t$ for all corrupted users $t \in T \cap C$.

- **Challenge phase:** When the adversary is finished making queries and proceeds to the challenge phase, the challenger starts by checking if $T = \emptyset$ or if $T \cap C \neq \emptyset$. In either case, the challenger halts the experiment with output 0. Otherwise, the challenger responds with the challenge ciphertext $\mathsf{ct}^* \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{apk}, b)$. The challenger also sets $T^* = T$ to remember the set of counters associated with the challenge ciphertext.

- **Refresh phase:** The challenger now performs a refresh. Specifically, the challenger first computes

$$(\mathsf{apk}', S', (\mathsf{ctref}, \rho)) = \mathsf{UpdateAPK}(\mathsf{pp}, \mathsf{apk}, (\perp, \perp), \mathsf{Refresh}, S).$$

  Then, for each $t \in T$, update

$$(\mathsf{ask}_t, S_t, \mathsf{pk}_t, \mathsf{sk}_t) \leftarrow \mathsf{UpdateASK}(\mathsf{pp}, (i_t, \mathsf{pk}_t, \mathsf{sk}_t, \mathsf{ask}_t), (\perp, \perp), \mathsf{ctref}, \mathsf{Refresh}, S).$$

  The challenger checks that $S_t = S'$ for each $t \in T$ and then it updates $\mathsf{D}[t] = (i_t, \mathsf{pk}_t, \mathsf{sk}_t, \mathsf{ask}_t)$, then sets $\mathsf{apk} = \mathsf{apk}'$ and $S = S'$. If the check fails, it halts the experiment and outputs 0. The challenger now responds to the adversary with the updated aggregate public key apk and the auxiliary update information ctref.

  Note that unlike the query phase, the challenger does not reveal the randomness $\rho$ used to compute the ciphertext ctref in the refresh phase.

- **Post-challenge query phase:** Same as the pre-challenge query phase above, except the adversary is no longer allowed to make a corruption query on any counter $c \in T^*$.

- **Reveal phase:** The challenger reveals the aggregated public key apk and the dictionary D to the adversary. Namely, for all $c \leq \mathsf{ctr}$, the challenger outputs $\mathsf{D}[c] = (i, \mathsf{pk}_c, \mathsf{sk}_c, \mathsf{ask}_c)$ to the adversary.

- **Output phase:** At the end of the experiment, the adversary outputs a bit $b' \in \{0,1\}$, which is the output of the experiment.

We say an updatable DBE scheme is *secure* if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\Big|\Pr[b' = 1 : b = 0] - \Pr[b' = 1 : b = 1]\Big| \leq \mathsf{negl}(\lambda),$$

where $b' \in \{0,1\}$ is the output of an execution of the above security experiment.

**Definition 5.9** (Static Security)**.** The *static* security game for an updatable DBE scheme $\Pi_{\mathsf{UpdDBE}}$ is identical to the security game from Definition 5.8, except the adversary commits to all of its pre-challenge queries (i.e., key-generation queries, corruption queries, and update queries) at the beginning of the experiment. The post-challenge query phase is unchanged. We say $\Pi_{\mathsf{UpdDBE}}$ satisfies *static* security if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\Big|\Pr[b' = 1 : b = 0] - \Pr[b' = 1 : b = 1]\Big| \leq \mathsf{negl}(\lambda),$$

where $b' \in \{0,1\}$ is the output of an execution of the static security experiment.

## 5.1 Construction of Updatable Distributed Broadcast Encryption

Our construction is a modification of the construction from Section 4.1 with two additional algorithms UpdPK and UpdSK. One crucial difference in the construction is that the secret key vector $\mathbf{r}_i$ is sampled uniformly from the range $\{0,\dots,\mathfrak{B}-1\}^{\hat{m}}$ (where the bound $\mathfrak{B}$ is set in the parameter section) instead of being sampled from $\{0,1\}^{\hat{m}}$, and our construction uses a hash function $H\colon \{0,1\}^{\lambda} \times \{0,1\}^{\lambda} \to \{0,1\}^{\hat{m}}$ that is modeled as a random oracle [BR93]. We color the primary changes from Section 4 in green.

**Construction 5.10** (Updatable Distributed Broadcast Encryption)**.** Let $\lambda$ be a security parameter and $(n, m, q, \widetilde{\sigma})$ be LWE parameters (which are functions of $\lambda$). Let $\sigma$ be a Gaussian width parameter and $\mathfrak{B}$ be a bound parameter. Let $\hat{m} = 4m^5$ and $\xi = \lfloor q/2 \rfloor$. We construct an updatable distributed broadcast encryption scheme $\Pi_{\mathsf{UpdDBE}} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{UpdPK}, \mathsf{UpdSK})$ as follows:

- $\mathsf{Setup}(1^{\lambda})$: On input the security parameter $\lambda$, the setup algorithm samples the matrix commitment parameters $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B}) \leftarrow \mathsf{Setup}_{\mathsf{com}}(1^{\lambda})$, together with $\mathbf{A} \xleftarrow{R} \mathbb{Z}_q^{n \times m}$ and $\mathbf{p} \xleftarrow{R} \mathbb{Z}_q^{n}$. It outputs $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$.
- $\mathsf{KeyGen}(\mathsf{pp}, i)$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$ where $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$ and an index $i \in [2^{\lambda}]$, the key-generation algorithm samples $\mathbf{r}_i \xleftarrow{R} \{0,\dots,\mathfrak{B}-1\}^{\hat{m}}$ and computes the verification vector $\mathbf{v}_i = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i)$. It then defines the public key $\mathbf{t}_i := \mathbf{B}\mathbf{r}_i + \mathbf{p} - \mathbf{A}\mathbf{v}_i \in \mathbb{Z}_q^n$ and the initial local opening $\widetilde{\mathbf{z}}_i := \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^{\intercal} \otimes \mathbf{t}_i, i)$, where $\mathbf{u}_i$ is the $i^{\text{th}}$ standard basis vector. It outputs the public key $\mathsf{pk}_i = \mathbf{t}_i$ and the secret key $\mathsf{sk}_i = (i, \mathbf{r}_i, \widetilde{\mathbf{z}}_i)$.
- $\mathsf{IncPK}(\mathsf{pp}, \mathsf{apk}, (i, \mathsf{pk}_i), \mathsf{op})$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, an aggregate public key $\mathsf{apk} = \widetilde{\mathbf{C}} \in \mathbb{Z}_q^{n \times m}$ (where we interpret $\mathsf{apk} = \perp$ to be the all-zeroes matrix), a public key $\mathsf{pk}_i = \mathbf{t}_i$ for user $i$, and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, the public-update algorithm computes the commitment

  $$\mathbf{C}_{(i)} = \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^{\intercal} \otimes \mathbf{t}_i). \tag{5.1}$$

  If $\mathsf{op} = \mathsf{Add}$, it outputs $\mathsf{apk}' = \widetilde{\mathbf{C}} + \mathbf{C}_{(i)}$. If $\mathsf{op} = \mathsf{Remove}$, it outputs $\mathsf{apk}' = \widetilde{\mathbf{C}} - \mathbf{C}_{(i)}$.
- $\mathsf{IncSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, (i, \mathsf{pk}_i), \mathsf{op})$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, an aggregate secret key $\mathsf{ask}_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*})$, a public key $\mathsf{pk}_i = \mathbf{t}_i$ for user $i$, and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, the private-update algorithm proceeds as follows:
  - If $i^* = i$, it returns $\mathsf{ask}_{i^*}$ unchanged. Otherwise, it computes the local opening

    $$\mathbf{z}_{(i,i^*)} = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^{\intercal} \otimes \mathbf{t}_i, i^*). \tag{5.2}$$

- If $\mathsf{op} = \mathsf{Add}$, it outputs $\mathsf{ask}'_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*} + \mathbf{z}_{(i,i^*)})$.
- If $\mathsf{op} = \mathsf{Remove}$, it outputs $\mathsf{ask}'_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*} - \mathbf{z}_{(i,i^*)})$.

- $\mathsf{Enc}(\mathsf{pp}, \mathsf{apk}, \mu)$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$ where $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$, an aggregate public key $\mathsf{apk} = \widetilde{\mathbf{C}}$, and a message $\mu \in \{0,1\}$, the encryption algorithm samples

$$\mathbf{s} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n, \quad \mathbf{e} \leftarrow \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\hat{m}}, \quad \mathbf{D}_1 \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m} \times m}, \quad \mathbf{d}_2 \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}.$$

  If $\|\mathbf{e}\| \geq \sqrt{\lambda}\,\widetilde{\sigma}$, it resets $\mathbf{e} = 0^{\hat{m}}$. It then outputs the ciphertext $\mathsf{ct} = (\mathbf{c}_1, \mathbf{c}_2, c_3)$ where

$$\mathbf{c}_1^\intercal = \mathbf{s}^\intercal \mathbf{B} + \mathbf{e}^\intercal, \quad \mathbf{c}_2^\intercal = \mathbf{s}^\intercal(\mathbf{A} + \widetilde{\mathbf{C}}) + \mathbf{e}^\intercal \mathbf{D}_1, \quad c_3 = \mathbf{s}^\intercal \mathbf{p} + \mathbf{e}^\intercal \mathbf{d}_2 + \xi \cdot \mu.$$

- $\mathsf{Dec}(\mathsf{pp}, \mathsf{ask}_{i^*}, \mathsf{ct})$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, an aggregate secret key $\mathsf{ask}_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*})$, and a ciphertext $\mathsf{ct} = (\mathbf{c}_1, \mathbf{c}_2, c_3)$, the decryption algorithm computes the verification vector $\mathbf{v}_{i^*} = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i^*)$ and outputs

$$\mu' = \lfloor c_3 + \mathbf{c}_1^\intercal(\mathbf{r}_{i^*} - \widetilde{\mathbf{z}}_{i^*}) - \mathbf{c}_2^\intercal \mathbf{v}_{i^*} \rceil_2.$$

- $\mathsf{UpdPK}(\mathsf{pp}, \mathsf{apk}, S)$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, an aggregate public key $\mathsf{apk} = \widetilde{\mathbf{C}}$, and a mapping $S$ from the index of each group member to their public key, the algorithm samples $\boldsymbol{\delta} \xleftarrow{\mathrm{R}} \{0,1\}^\lambda$ and for all $i \in S$, sets $\boldsymbol{\Delta}_i = H(\boldsymbol{\delta}, i)$. For all $j \in [\lambda]$, it computes $\mathsf{ctref}_{(j)} \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{apk}, \delta_j)$ where $\delta_j$ is the $j^{\text{th}}$ component of vector $\boldsymbol{\delta}$. Let $\mathsf{ctref} = \big(\mathsf{ctref}_{(1)}, \ldots, \mathsf{ctref}_{(\lambda)}\big)$. Then, for each $i \in S$, let $\mathbf{t}_i = S[i]$ and set $S'[i] = \mathbf{t}_i + \mathbf{B}\boldsymbol{\Delta}_i$. Compute

$$\mathsf{apk}' = \sum_{i \in S} \mathsf{Com}\left(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes (\mathbf{t}_i + \mathbf{B}\boldsymbol{\Delta}_i)\right)$$

  and output $(\mathsf{ctref}, \mathsf{apk}', S')$.

- $\mathsf{UpdSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, \mathsf{ctref}, S, \mathsf{pk}_{i^*}, \mathsf{sk}_{i^*})$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, an aggregate secret key $\mathsf{ask}_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*})$, a ciphertext $\mathsf{ctref} = \big(\mathsf{ctref}_{(1)}, \ldots, \mathsf{ctref}_{(\lambda)}\big)$ where $\mathsf{ctref}_{(j)} = \big(\mathbf{c}_{1,(j)}, \mathbf{c}_{2,(j)}, c_{3,(j)}\big)$, a mapping $S$ of user indices to public keys, and a public key $\mathsf{pk}_{i^*} = \mathbf{t}_{i^*}$ together with a secret key $\mathsf{sk}_{i^*} = \big(i^*, \mathbf{r}_{i^*}, \mathbf{z}_{(i^*,i^*)}\big)$ for user $i^*$, the private-refresh algorithm proceeds as follows. Compute the verification vector $\mathbf{v}_{i^*} = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i^*)$ and, for each $j \in [\lambda]$,

$$\delta_j = \lfloor c_{3,(j)} + \mathbf{c}_{1,(j)}^\intercal(\mathbf{r}_{i^*} - \widetilde{\mathbf{z}}_{i^*}) - \mathbf{c}_{2,(j)}^\intercal \mathbf{v}_{i^*} \rceil_2,$$

  and set $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_\lambda) \in \{0,1\}^\lambda$. For each $i \in S$, write $\mathbf{t}_i = S[i]$ (in particular $\mathbf{t}_{i^*} = S[i^*] = \mathsf{pk}_{i^*}$), compute $\boldsymbol{\Delta}_i = H(\boldsymbol{\delta}, i)$, and set

$$\mathbf{t}'_i \coloneqq \mathbf{t}_i + \mathbf{B}\boldsymbol{\Delta}_i \quad \text{and} \quad S'[i] \coloneqq \mathbf{t}'_i.$$

  Output $(\mathsf{ask}'_{i^*}, \mathsf{pk}'_{i^*}, \mathsf{sk}'_{i^*}, S')$, where

$$\begin{aligned}
\mathsf{pk}'_{i^*} &= \mathbf{t}'_{i^*}, \\
\mathsf{ask}'_{i^*} &= \left(i^*,\ \mathbf{r}_{i^*} + \boldsymbol{\Delta}_{i^*},\ \sum_{i \in S} \mathsf{Open}\left(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes \mathbf{t}'_i,\ i^*\right)\right), \\
\mathsf{sk}'_{i^*} &= \left(i^*,\ \mathbf{r}_{i^*} + \boldsymbol{\Delta}_{i^*},\ \mathsf{Open}\left(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i^*}^\intercal \otimes \mathbf{t}'_{i^*},\ i^*\right)\right).
\end{aligned}$$

**Theorem 5.11** (Correctness)**.** Suppose $n, m, \sigma, \widetilde{\sigma} = \mathsf{poly}(\lambda)$, $\mathfrak{B} \leq 2^\lambda$ and $q/\log q > 2^\lambda \cdot \mathsf{poly}(\lambda)$ where $\mathsf{poly}(\lambda)$ is the specific polynomial produced in the noise analysis. Then, Construction 5.10 is correct.

*Proof.* Fix a security parameter $\lambda$, a set of public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$ in the support of $\mathsf{Setup}(1^\lambda)$, an index $i^* \in [2^\lambda]$, a key-pair $(\mathsf{pk}_{i^*}, \mathsf{sk}_{i^*})$ in the support of $\mathsf{KeyGen}(\mathsf{pp}, i^*)$, and a sequence of queries $\{(\mathsf{ind}_j, \mathsf{pk}_j, \mathsf{op}_j)\}_{j \in [Q]}$ where $Q \leq 2^\lambda$. For $j \in \{0, 1, \ldots, Q\}$, let $(\mathsf{apk}_j, \mathsf{ask}_{i^*,j}, \mathsf{pk}_{i^*,j}, \mathsf{sk}_{i^*,j}, S_j)$ be the values computed in Definition 5.4. Our goal is to show that if $S_Q[i^*] = \mathsf{pk}_{i^*,Q}$, then mapping consistency and decryption correctness hold. Formally, for all $j \in [Q]$ such that $i^* \in S_j$, we have $S_{i^*,j} = S_j$ and for all messages $\mu \in \{0,1\}$, we have $\mathsf{Dec}(\mathsf{pp}, \mathsf{ask}_{i^*,Q}, \mathsf{ct}) = \mu$ with probability 1, where $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{apk}_Q, \mu)$. From Fact 3.5, we have $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$, where $\|\mathbf{R}\| \leq \sigma\sqrt{\lambda}$. Moreover, for all matrices $\mathbf{M} \in \mathbb{Z}_q^{n \times L}$ with $L \leq 2^\lambda$ and all indices $i \in [2^\lambda]$, the vectors $\mathbf{v}_i = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i)$ and $\mathbf{z}_i = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{M}, i)$ satisfy

$$\|\mathbf{v}_i\| \leq O\big(m^4 \sigma \log q \cdot \lambda^{1/2}\big) \qquad \text{and} \qquad \|\mathbf{z}_i\| \leq O\big(m^7 \sigma \log q \cdot \lambda^{3/2}\big). \tag{5.3}$$

**Induction invariant.** For all $j \in \{0, 1, \ldots, Q\}$, and for every user $j' \in S_j$, let the public key of user with index $j'$ after operation $j$ be denoted by $\mathbf{t}_{j',j} = S_j[j'] \in \mathbb{Z}_q^n$. For all $j \in \{0, 1, \ldots, Q\}$, and for any user $i^* \in S_j$ we prove correctness by proving an induction invariant on the tuple $(\mathsf{apk}_j, \mathsf{ask}_{i^*,j}, \mathsf{pk}_{i^*,j}, \mathsf{sk}_{i^*,j}, S_j)$, together with the mapping $S_{i^*,j}$ computed by user $i^*$ via UpdateASK (Definition 5.3):

- **Mapping consistency:** if $i^* \in S_j$, then $S_{i^*,j} = S_j$. In particular, for every index $j' \in S_j$, $S_{i^*,j}[j'] = S_j[j'] = \mathbf{t}_{j',j}$.
- **Aggregated public key:** For any user $j' \in S_j$, let $\mathbf{C}_{(j')}$ be the commitment associated with $(j', \mathbf{t}_{j',j})$ from Eq. (5.1). We have $\mathsf{apk}_j = \sum_{j' \in S_j} \mathbf{C}_{(j')}$, where we interpret $\mathsf{apk}_j = \perp$ as the all-zeroes matrix.
- **Aggregated secret key:** Let $\mathbf{z}_{(j',i^*)}$ be the local opening associated with $(j', \mathbf{t}_{j',j})$ from Eq. (5.2). If $S_j[i^*] = \mathsf{pk}_{i^*,j}$, then $\mathsf{ask}_{i^*,j} = (i^*, \mathbf{r}_{i^*,j}, \widetilde{\mathbf{z}}_{i^*,j})$ where $\widetilde{\mathbf{z}}_{i^*,j} = \sum_{j' \in S_j} \mathbf{z}_{(j',i^*)}$.
- **Public/secret key:** if $S_j[i^*] = \mathsf{pk}_{i^*,j}$, then $\mathsf{sk}_{i^*,j} = (i^*, \mathbf{r}_{i^*,j}, \mathbf{z}_{(i^*,i^*)})$ where $\mathbf{r}_{i^*,j} \in \{0, \ldots, \mathfrak{B} + j - 1\}^{\hat{m}}$ matches the stored vector in $\mathsf{ask}_{i^*,j}$. Additionally, $\mathbf{z}_{(i^*,i^*)}$ satisfies Eq. (5.2) and $\mathsf{pk}_{i^*,j} = \mathbf{t}_{i^*,j} = \mathbf{B}\mathbf{r}_{i^*,j} + \mathbf{p} - \mathbf{A}\mathbf{v}_{i^*}$ where $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, $\mathbf{v}_{i^*} = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i^*)$.

Before we prove the invariant, we prove the following important claim:

**Claim 5.12.** Fix $j \in [Q]$ and suppose the induction invariant holds for $(\mathsf{apk}_{j-1}, \mathsf{ask}_{i^*,j-1}, \mathsf{pk}_{i^*,j-1}, \mathsf{sk}_{i^*,j-1}, S_{j-1})$ and that $i^* \in S_{j-1}$ with $S_{j-1}[i^*] = \mathsf{pk}_{i^*,j-1}$ (so that the aggregated secret key and public/secret key invariants apply to user $i^*$). Let $(\mathsf{ctref}, \mathsf{apk}_j, S_j) \leftarrow \mathsf{UpdPK}(\mathsf{pp}, \mathsf{apk}_{j-1}, S_{j-1})$ and let $\boldsymbol{\delta} \in \{0,1\}^\lambda$ be the seed sampled inside UpdPK. Then the vector $\boldsymbol{\delta}'$ recovered by $\mathsf{UpdSK}(\mathsf{pp}, \mathsf{ask}_{i^*,j-1}, \mathsf{ctref}, S_{j-1}, \mathsf{pk}_{i^*,j-1}, \mathsf{sk}_{i^*,j-1})$ satisfies $\boldsymbol{\delta}' = \boldsymbol{\delta}$.

*Proof.* Let the public key for user with index $j'$ after operation $j-1$ be denoted by $\mathbf{t}_{j',j-1} = S_{j-1}[j']$, and write $\widetilde{\mathbf{C}} = \mathsf{apk}_{j-1}$, $\mathsf{ask}_{i^*,j-1} = (i^*, \mathbf{r}_{i^*,j-1}, \widetilde{\mathbf{z}}_{i^*,j-1})$ and $\mathsf{sk}_{i^*,j-1} = (i^*, \mathbf{r}_{i^*,j-1}, \mathbf{z}_{(i^*,i^*)})$. Since $S_{j-1}[i^*] = \mathsf{pk}_{i^*,j-1}$, the induction invariant gives

$$\widetilde{\mathbf{C}} = \sum_{j' \in S_{j-1}} \mathbf{C}_{(j')} \qquad \text{and} \qquad \widetilde{\mathbf{z}}_{i^*,j-1} = \sum_{j' \in S_{j-1}} \mathbf{z}_{(j',i^*)}.$$

From Eqs. (5.1) and (5.2), we have that $\mathbf{C}_{(j')} = \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{j'}^\mathsf{T} \otimes \mathbf{t}_{j',j-1})$ and $\mathbf{z}_{(j',i^*)} = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{j'}^\mathsf{T} \otimes \mathbf{t}_{j',j-1}, i^*)$. By Claim 4.10, this means

$$\widetilde{\mathbf{C}}\mathbf{v}_{i^*} = \sum_{j' \in S_{j-1}} \mathbf{C}_{(j')} \cdot \mathbf{v}_{i^*} = \mathbf{t}_{i^*,j-1} - \mathbf{B} \sum_{j' \in S_{j-1}} \mathbf{z}_{(j',i^*)} = \mathbf{t}_{i^*,j-1} - \mathbf{B}\widetilde{\mathbf{z}}_{i^*,j-1}. \tag{5.4}$$

Moreover, by Eq. (5.3),

$$\|\widetilde{\mathbf{z}}_{i^*,j-1}\| \le |S_{j-1}| \cdot O\big(m^7 \sigma \log q \cdot \lambda^{3/2}\big). \tag{5.5}$$

Fix an index $\ell \in [\lambda]$ and let $\delta_\ell \in \{0,1\}$ be the $\ell^{\text{th}}$ coordinate of $\boldsymbol{\delta}$. Let $\mathsf{ctref}_{(\ell)} = (\mathbf{c}_{1,(\ell)}, \mathbf{c}_{2,(\ell)}, c_{3,(\ell)})$ be the corresponding encryption of $\delta_\ell$. By construction of ctref in Construction 5.10, we can write

$$\mathbf{c}_{1,(\ell)}^\mathsf{T} = \mathbf{s}^\mathsf{T}\mathbf{B} + \mathbf{e}^\mathsf{T}, \quad \mathbf{c}_{2,(\ell)}^\mathsf{T} = \mathbf{s}^\mathsf{T}(\mathbf{A} + \widetilde{\mathbf{C}}) + \mathbf{e}^\mathsf{T}\mathbf{D}_1, \quad c_{3,(\ell)} = \mathbf{s}^\mathsf{T}\mathbf{p} + \mathbf{e}^\mathsf{T}\mathbf{d}_2 + \xi \cdot \delta_\ell.$$

The UpdSK algorithm computes

$$\delta'_\ell = \lfloor c_{3,(\ell)} + \mathbf{c}_{1,(\ell)}^\mathsf{T}(\mathbf{r}_{i^*,j-1} - \widetilde{\mathbf{z}}_{i^*,j-1}) - \mathbf{c}_{2,(\ell)}^\mathsf{T}\mathbf{v}_{i^*} \rceil_2.$$

By the invariant condition, $\mathbf{p} + \mathbf{B}\mathbf{r}_{i^*,j-1} - \mathbf{A}\mathbf{v}_{i^*} = \mathbf{t}_{i^*,j-1}$, and by Eq. (5.4), $\widetilde{\mathbf{C}}\mathbf{v}_{i^*} = \mathbf{t}_{i^*,j-1} - \mathbf{B}\widetilde{\mathbf{z}}_{i^*,j-1}$, so we have

$$\delta'_\ell = \lfloor \xi\, \delta_\ell + \underbrace{\mathbf{e}^\mathsf{T}\mathbf{d}_2 + \mathbf{e}^\mathsf{T}\mathbf{r}_{i^*,j-1} - \mathbf{e}^\mathsf{T}\mathbf{D}_1\mathbf{v}_{i^*} - \mathbf{e}^\mathsf{T}\widetilde{\mathbf{z}}_{i^*,j-1}}_{\nu} \rceil_2.$$

It remains to bound $|\nu|$. By construction of ctref, $\|\mathbf{e}\| \le \lambda^{1/2}\widetilde{\sigma}$. In addition, $\mathbf{d}_2 \in \{0,1\}^{\hat{m}}$, $\mathbf{D}_1 \in \{0,1\}^{\hat{m} \times m}$ and by the invariant, the norm bound $\|\mathbf{r}_{i^*,j-1}\| \le \mathfrak{B} + j - 2$ holds. Thus, by Eq. (5.3) and Eq. (5.5) and the bound $\mathfrak{B} \le 2^\lambda$, this gives

$$|\nu| \le \hat{m}\,\lambda^{1/2}\widetilde{\sigma}\left(1 + (\mathfrak{B} + j - 2) + O\big(m^5 \sigma \log q \cdot \lambda^{1/2}\big) + |S_{j-1}| \cdot O\big(m^7 \sigma \log q \cdot \lambda^{3/2}\big)\right) \le O\big(2^\lambda \cdot \hat{m} m^7 \sigma\widetilde{\sigma} \cdot \lambda^2 \log q\big),$$

where the second inequality uses $|S_{j-1}| \geq 1$ (which holds since $i^* \in S_{j-1}$), $|S_{j-1}| \leq (j-1) \leq 2^\lambda$, and $\mathfrak{B} + j - 2 \leq 2^{\lambda+1}$. We stress that in contrast to Theorem 4.12, the secret-key range $\mathfrak{B}$ (and not just the group size $|S_{j-1}|$) now feeds into the noise bound. Finally, since $n, m, \sigma, \widetilde{\sigma} = \mathsf{poly}(\lambda)$, we conclude that

$$|\nu| \leq 2^\lambda \cdot \mathsf{poly}(\lambda) \cdot \log q.$$

Setting $q/\log q > 4 \cdot 2^\lambda \cdot \mathsf{poly}(\lambda)$ means $|\nu| < q/4$. Since $\xi = \lfloor q/2 \rfloor$, the rounding operation recovers $\delta'_\ell = \delta_\ell$. As this holds for every $\ell \in [\lambda]$, we conclude that $\boldsymbol{\delta}' = \boldsymbol{\delta}$. □

**Claim 5.13.** For all $j \in \{0, 1, \ldots, Q\}$, the induction invariant holds.

*Proof.* We proceed by induction on $j$. Claim 5.12 is stated conditionally on the invariant at step $j-1$; together with this claim it forms a single induction, with Claim 5.12 supplying the Refresh case below.

**Base case.** For the base case where $j = 0$, we have $\mathsf{apk}_0 = \bot$, $\mathsf{ask}_{i^*,0} = \bot$, $S_0 = \varnothing$, and $S_{i^*,0} = \varnothing$. All of the properties hold trivially since $S_0 = \varnothing$.

**Inductive step.** For the inductive step, suppose the invariant holds for $(\mathsf{apk}_{j-1}, \mathsf{ask}_{i^*,j-1}, \mathsf{pk}_{i^*,j-1}, \mathsf{sk}_{i^*,j-1}, S_{j-1})$ and consider the query $(\mathsf{ind}_j, \mathsf{pk}_j, \mathsf{op}_j)$. To simplify notation, let $i = \mathsf{ind}_j$. First, if $\mathsf{op}_j \in \{\mathsf{Add}, \mathsf{Remove}\}$ and neither of the conditions ($\mathsf{op}_j = \mathsf{Add}$ and $i \notin S_{j-1}$) or ($\mathsf{op}_j = \mathsf{Remove}$ and $i \in S_{j-1}$) holds, then UpdateAPK and UpdateASK leave their inputs unchanged. In this case,

$$\mathsf{apk}_j = \mathsf{apk}_{j-1}, \qquad \mathsf{ask}_{i^*,j} = \mathsf{ask}_{i^*,j-1}, \qquad \mathsf{pk}_{i^*,j} = \mathsf{pk}_{i^*,j-1}, \qquad \mathsf{sk}_{i^*,j} = \mathsf{sk}_{i^*,j-1} \quad \text{and} \quad S_j = S_{j-1}.$$

Thus, if the induction invariant holds for $(\mathsf{apk}_{j-1}, \mathsf{ask}_{i^*,j-1}, \mathsf{pk}_{i^*,j-1}, \mathsf{sk}_{i^*,j-1}, S_{j-1})$, then it also holds for

$$(\mathsf{apk}_j, \mathsf{ask}_{i^*,j}, \mathsf{pk}_{i^*,j}, \mathsf{sk}_{i^*,j}, S_j).$$

It remains to consider the three cases where the query is non-trivial:

- **Case $\mathsf{op}_j = \mathsf{Add}$ and $i \notin S_{j-1}$:** In this case, $S_j = S_{j-1} \cup \{i\}$ with $S_j[i] = \mathsf{pk}_j$ and $S_j[j'] = S_{j-1}[j']$ for all $j' \in S_{j-1}$. By construction, IncPK outputs

$$\mathsf{apk}_j = \mathsf{apk}_{j-1} + \mathbf{C}_{(i)} = \sum_{j' \in S_{j-1}} \mathbf{C}_{(j')} + \mathbf{C}_{(i)} = \sum_{j' \in S_j} \mathbf{C}_{(j')},$$

  where the second equality uses the inductive hypothesis. For the aggregated secret key, suppose $S_j[i^*] = \mathsf{pk}_{i^*,j}$ (otherwise, the second property holds vacuously). We consider two sub-cases:

  - Suppose $i^* \neq i$. Then $S_{j-1}[i^*] = S_j[i^*] = \mathsf{pk}_{i^*,j} = \mathsf{pk}_{i^*,j-1}$. Then, by the inductive hypothesis, we have that $\mathsf{ask}_{i^*,j-1} = (i^*, \mathbf{r}_{i^*,j-1}, \sum_{j' \in S_{j-1}} \mathbf{z}_{(j',i^*)})$. In this case, UpdateASK invokes IncSK on $(i, \mathsf{pk}_j)$ with operation Add, which adds the local opening $\mathbf{z}_{(i,i^*)}$ to the aggregated opening. Thus,

$$\widetilde{\mathbf{z}}_{i^*,j} = \sum_{j' \in S_{j-1}} \mathbf{z}_{(j',i^*)} + \mathbf{z}_{(i,i^*)} = \sum_{j' \in S_j} \mathbf{z}_{(j',i^*)}.$$

  - Suppose $i^* = i$. In this case, $\mathsf{pk}_j = S_j[i^*] = \mathsf{pk}_{i^*,j} = \mathsf{pk}_{i^*,j-1}$. Then UpdateASK initializes $\mathsf{ask}_{i^*,j} = \mathsf{sk}_{i^*,j-1} = (i^*, \mathbf{r}_{i^*,j-1}, \mathbf{z}_{(i^*,i^*)})$ and invokes IncSK on $(j', \mathbf{t}_{j',j-1})$ with operation Add for each member $j' \in S_{j-1}$, which adds the local opening $\mathbf{z}_{(j',i^*)}$. Thus,

$$\widetilde{\mathbf{z}}_{i^*,j} = \mathbf{z}_{(i^*,i^*)} + \sum_{j' \in S_{j-1}} \mathbf{z}_{(j',i^*)} = \sum_{j' \in S_j} \mathbf{z}_{(j',i^*)}.$$

  In both cases, we do not update the public/secret keys for any user. We also add $\mathsf{pk}_j$ to index $i$ in the mapping set. Since UpdateASK initializes $S_{i^*,j} = S_{j-1}$ and then sets $S_{i^*,j}[i] = \mathsf{pk}_j$, we have $S_{i^*,j} = S_j$, so mapping consistency holds. From the induction hypothesis and adding index $i$ to set $S_{j-1}$, we can conclude that the invariant holds.

- **Case** $\mathrm{op}_j = \mathsf{Remove}$ **and** $i \in S_{j-1}$**:** In this case, $S_j = S_{j-1} \setminus \{i\}$. By the inductive hypothesis, $S_{j-1}[i] = \mathbf{t}_{i,j-1}$, and by construction IncPK subtracts the commitment $\mathbf{C}_{(i)} = \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\top \otimes S_{j-1}[i]) = \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\top \otimes \mathbf{t}_{i,j-1})$, which by the inductive hypothesis is exactly the term contributed by user $i$ to $\mathsf{apk}_{j-1}$. Thus, the output of IncPK satisfies
$$\mathsf{apk}_j = \mathsf{apk}_{j-1} - \mathbf{C}_{(i)} = \sum_{j' \in S_{j-1}} \mathbf{C}_{(j')} - \mathbf{C}_{(i)} = \sum_{j' \in S_j} \mathbf{C}_{(j')}.$$
For the aggregated secret key, suppose $S_j[i^*] = \mathsf{pk}_{i^*,j}$ (otherwise the second property holds vacuously). This means that $i^* \neq i$ and $S_{j-1}[i^*] = \mathsf{pk}_{i^*,j-1}$, so by the inductive hypothesis, $\mathsf{ask}_{i^*,j-1} = (i^*, \mathbf{r}_{i^*,j-1}, \sum_{j' \in S_{j-1}} \mathbf{z}_{(j',i^*)})$. In this case, UpdateASK invokes IncSK on $(i, S_{j-1}[i])$ with operation Remove, which subtracts the local opening $\mathbf{z}_{(i,i^*)}$. Since Open is deterministic, this means
$$\widetilde{\mathbf{z}}_{i^*,j} = \sum_{j' \in S_{j-1}} \mathbf{z}_{(j',i^*)} - \mathbf{z}_{(i,i^*)} = \sum_{j' \in S_j} \mathbf{z}_{(j',i^*)}.$$
We do not update the public/secret keys for any user. We also remove index $i$ from the mapping set. Since UpdateASK initializes $S_{i^*,j} = S_{j-1}$ and then removes index $i$—the identical rule UpdateAPK uses to update $S_{j-1}$ to $S_j$—we have $S_{i^*,j} = S_j$, so mapping consistency holds. From the induction hypothesis and removing index $i$ from set $S_{j-1}$, we can conclude that the invariant holds.

- **Case** $\mathrm{op}_j = \mathsf{Refresh}$**:** In this case, we first observe that the index sets of $S_{j-1}$ and $S_j$ coincide (i.e., no user is added or removed); only the public keys they store change. Throughout this case, we assume that $i^* \in S_{j-1}$ and $S_{j-1}[i^*] = \mathsf{pk}_{i^*,j-1}$; if $i^* \notin S_{j-1}$, then $\mathsf{ask}_{i^*,j}, \mathsf{pk}_{i^*,j}, \mathsf{sk}_{i^*,j}$ are unchanged and the aggregated secret key and public/secret key invariants hold vacuously (since $S_j[i^*]$ is undefined), while the mapping-consistency requirement is likewise vacuous.

  In a refresh operation, the UpdateAPK procedure computes a ciphertext $\mathsf{ctref} = (\mathsf{ctref}_{(1)}, \ldots, \mathsf{ctref}_{(\lambda)})$ where $\mathsf{ctref}_{(\ell)}$ is an encryption of the $\ell^{\text{th}}$ bit $\delta_\ell$ of the seed $\boldsymbol{\delta}$ under the aggregate public key $\mathsf{apk}_{j-1}$. In the UpdateASK procedure, user $i^*$ uses $\mathsf{ask}_{i^*,j-1}$ to decrypt ctref and recovers a vector $\boldsymbol{\delta}'$. Since the induction hypothesis holds for $(\mathsf{apk}_{j-1}, \mathsf{ask}_{i^*,j-1}, \mathsf{pk}_{i^*,j-1}, \mathsf{sk}_{i^*,j-1}, S_{j-1})$ and $i^* \in S_{j-1}$ with $S_{j-1}[i^*] = \mathsf{pk}_{i^*,j-1}$, Claim 5.12 applies and user $i^*$ recovers exactly the seed $\boldsymbol{\delta}$ sampled by UpdPK.

  Consequently, for every $j' \in S_{j-1}$, user $i^*$ computes the identical shift $\boldsymbol{\Delta}_{j'} = H(\boldsymbol{\delta}, j')$ used by UpdPK, so $S_{i^*,j}[j'] = \mathbf{t}_{j',j-1} + \mathbf{B}\boldsymbol{\Delta}_{j'} = S_j[j']$ for every $j'$. This establishes mapping consistency for this step.

  By construction, UpdPK computes the aggregated public key $\mathsf{apk}_j = \sum_{j' \in S_j} \mathbf{C}_{(j')}$ where $\mathbf{C}_{(j')} = \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{j'}^\top \otimes \mathbf{t}_{j',j})$ and $\mathbf{t}_{j',j} = \mathbf{t}_{j',j-1} + \mathbf{B}\boldsymbol{\Delta}_{j'}$, so the aggregated public key invariant is maintained.

  Similarly, for the aggregated secret key, UpdSK computes $\mathsf{ask}_{i^*,j} = (i^*, \mathbf{r}_{i^*,j-1} + \boldsymbol{\Delta}_{i^*}, \widetilde{\mathbf{z}}_{i^*,j})$ where $\widetilde{\mathbf{z}}_{i^*,j} = \sum_{j' \in S_j} \mathbf{z}_{(j',i^*)}$ and $\mathbf{z}_{(j',i^*)}$ is the local opening at position $i^*$ associated with $(j', \mathbf{t}_{j',j})$. By construction, the aggregated secret key invariant is maintained.

  Finally, the secret key update consistently stores $\mathbf{r}_{i^*,j} = \mathbf{r}_{i^*,j-1} + \boldsymbol{\Delta}_{i^*}$ in $\mathsf{ask}_{i^*,j}$ and $\mathsf{sk}_{i^*,j}$. Since the induction hypothesis holds, $\mathbf{t}_{i^*,j-1} = \mathbf{B}\mathbf{r}_{i^*,j-1} + \mathbf{p} - \mathbf{A}\mathbf{v}_{i^*}$ where $\mathbf{r}_{i^*,j-1} \in \{0, \ldots, \mathfrak{B} + j - 2\}^{\hat{m}}$. Adding $\mathbf{B}\boldsymbol{\Delta}_{i^*}$ to this equation, we get $\mathbf{t}_{i^*,j} = \mathbf{t}_{i^*,j-1} + \mathbf{B}\boldsymbol{\Delta}_{i^*} = \mathbf{B}(\mathbf{r}_{i^*,j-1} + \boldsymbol{\Delta}_{i^*}) + \mathbf{p} - \mathbf{A}\mathbf{v}_{i^*}$. Since $\boldsymbol{\Delta}_{i^*}$ is a binary vector, each entry of $\mathbf{r}_{i^*,j}$ grows by at most one, so $\mathbf{r}_{i^*,j} \in \{0, \ldots, \mathfrak{B} + j - 1\}^{\hat{m}}$ as required, and UpdSK computes $\mathbf{z}_{(i^*,i^*)}$ satisfying Eq. (5.2). Finally, we set $\mathsf{pk}_{i^*,j} = \mathbf{t}_{i^*,j}$, maintaining all conditions of the public/secret key invariant.

In all three cases, we see that the induction invariant holds. The claim now follows by induction. □

**Completing the proof.** Claim 5.13 establishes mapping consistency. For decryption correctness, suppose $S_Q[i^*] = \mathsf{pk}_{i^*,Q}$ and let $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{apk}_Q, \mu)$. By the invariant at step $Q$, $\mathsf{apk}_Q = \sum_{j' \in S_Q} \mathbf{C}_{(j')}$ and $\mathsf{ask}_{i^*,Q} = (i^*, \mathbf{r}_{i^*,Q}, \widetilde{\mathbf{z}}_{i^*,Q})$ with $\widetilde{\mathbf{z}}_{i^*,Q} = \sum_{j' \in S_Q} \mathbf{z}_{(j',i^*)}$ and $\mathbf{t}_{i^*,Q} = \mathbf{B}\mathbf{r}_{i^*,Q} + \mathbf{p} - \mathbf{A}\mathbf{v}_{i^*}$, where $\mathbf{r}_{i^*,Q} \in \{0, \ldots, \mathfrak{B} + Q - 1\}^{\hat{m}}$. The computation in the proof of Claim 5.12 applies verbatim with $(\mathsf{ct}, \mu)$ in place of $(\mathsf{ctref}_{(\ell)}, \delta_\ell)$ and $S_Q$ in place of $S_{j-1}$, and the same bound on $|\nu|$ gives $|\nu| < q/4$. Hence $\mathsf{Dec}(\mathsf{pp}, \mathsf{ask}_{i^*,Q}, \mathsf{ct}) = \mu$ with probability 1. □

**Theorem 5.14** (Succinctness)**.** Suppose $n, m = \mathsf{poly}(\lambda)$ and $q \le 2^{\mathsf{poly}(\lambda)}$. Then, Construction 5.10 satisfies succinctness (Definition 5.6).

*Proof.* Take any $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$ in the support of $\mathsf{Setup}(1^\lambda)$. By Definition 5.6, it suffices to bound the size of the keys $(\mathsf{apk}, \mathsf{ask}_{i^*}, \mathsf{sk}_{i^*})$, along with the size of each individual public key in the output mapping, output by the procedures $\mathsf{IncPK}(\mathsf{pp}, \cdot, \cdot, \cdot)$, $\mathsf{IncSK}(\mathsf{pp}, \cdot, \cdot, \cdot)$, $\mathsf{UpdPK}(\mathsf{pp}, \cdot, \cdot)$, $\mathsf{UpdSK}(\mathsf{pp}, \cdot, \cdot, \cdot, \cdot, \cdot)$ by a universal polynomial in $\lambda$:

- Algorithm IncPK outputs a matrix $\mathsf{apk}' \in \mathbb{Z}_q^{n \times m}$, which can be represented using $nm\lceil \log q \rceil$ bits.
- Algorithm IncSK outputs a triple $\mathsf{ask}'_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*})$, where $i^* \in [2^\lambda]$ and $\mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*} \in \mathbb{Z}_q^{\hat{m}}$. These can be represented using $\lambda + 2\hat{m}\lceil \log q \rceil$ bits.
- Algorithm UpdPK outputs a matrix $\mathsf{apk}' \in \mathbb{Z}_q^{n \times m}$, a refresh ciphertext $\mathsf{ctref} = (\mathsf{ctref}_{(1)}, \ldots, \mathsf{ctref}_{(\lambda)})$ where each $\mathsf{ctref}_{(\ell)} = (\mathbf{c}_{1,(\ell)}, \mathbf{c}_{2,(\ell)}, c_{3,(\ell)}) \in \mathbb{Z}_q^{\hat{m}} \times \mathbb{Z}_q^m \times \mathbb{Z}_q$, and a mapping $S'$ in which each public key satisfies $S'[i] \in \mathbb{Z}_q^n$. The key $\mathsf{apk}'$ and ciphertext ctref can be represented by $nm\lceil \log q \rceil + \lambda \cdot \hat{m}\lceil \log q \rceil + \lambda \cdot m\lceil \log q \rceil + \lambda \cdot \lceil \log q \rceil$ bits, and each public key in $S'$ by $n\lceil \log q \rceil$ bits.
- Algorithm UpdSK outputs $\mathsf{ask}'_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*})$, where $i^* \in [2^\lambda]$ and $\mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*} \in \mathbb{Z}_q^{\hat{m}}$, a public key $\mathsf{pk}'_{i^*} \in \mathbb{Z}_q^n$, and a secret key $\mathsf{sk}'_{i^*}$ of the same form, and a mapping $S'$ in which each public key satisfies $S'[i] \in \mathbb{Z}_q^n$. The keys $\mathsf{ask}'_{i^*}$ and $\mathsf{sk}'_{i^*}$ can each be represented by $\lambda + 2\hat{m}\lceil \log q \rceil$ bits, $\mathsf{pk}'_{i^*}$ by $n\lceil \log q \rceil$ bits, and each public key in $S'$ by $n\lceil \log q \rceil$ bits.

When $n, m = \mathsf{poly}(\lambda)$, $\hat{m} = 4m^5$, and $q \le 2^{\mathsf{poly}(\lambda)}$, these outputs have size $\mathsf{poly}(\lambda)$, as required. □

**Theorem 5.15** (Security)**.** If the matrix commitment scheme is secure (Fact 3.5), $n \ge \lambda$, $m \ge 2(n+1)\log q$, $\sigma \ge \log m$, $q$ is prime, $\mathfrak{B} \in 2^{\omega(\log \lambda)}$, and the conditions in Theorem 5.11 hold, the construction above satisfies static security (Definition 5.9) in the random oracle model.

*Proof.* To prove security, we proceed by defining a sequence of hybrid experiments, each parameterized by a bit $b \in \{0,1\}$ and implicitly by an adversary $\mathcal{A}$ and a security parameter $\lambda$:

- $\mathsf{Hyb}_0^{(b)}$: This is the static security experiment with challenge bit $b \in \{0,1\}$. Recall that in the static security experiment, the adversary commits to all of its pre-challenge queries at the beginning of the experiment, before seeing the scheme parameters. Concretely, the experiment proceeds as follows:
  - **Static commitment phase:** At the beginning of the experiment, the adversary $\mathcal{A}$ commits to the entire sequence of pre-challenge queries it will make, where each query is one of the following:
    - a key-generation query on an index $i \in [2^\lambda]$;
    - a corruption query on a counter $c$; or
    - an update query on a counter $c$ and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{Refresh}\}$.
  - **Setup phase:** The challenger begins by sampling the public parameters
    $$\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B}) \leftarrow \mathsf{Setup}_{\mathsf{com}}(1^\lambda), \qquad \mathbf{A} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n \times m}, \qquad \mathbf{p} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n.$$
    Next, the challenger initializes a counter $\mathsf{ctr} = 0$, a dictionary $\mathsf{D} = \varnothing$, a mapping $S = \varnothing$, a set $T = \varnothing$, the aggregated public key $\widetilde{\mathbf{C}} = \mathbf{0}^{n \times m}$ (representing $\mathsf{apk} = \perp$), and a set of corrupted counters $C = \varnothing$. The challenger processes each of the committed queries in order and constructs the response to each query as follows:
    - **Key-generation query on an index** $i \in [2^\lambda]$**:** The challenger increments the counter $\mathsf{ctr} = \mathsf{ctr} + 1$. It samples $\mathbf{r}_{\mathsf{ctr}} \xleftarrow{\mathrm{R}} \{0, \ldots, \mathfrak{B} - 1\}^{\hat{m}}$ and computes
      $$\mathbf{v}_i = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i) \in \mathbb{Z}_q^m \qquad \text{and} \qquad \mathbf{t}_{\mathsf{ctr}} = \mathbf{B}\mathbf{r}_{\mathsf{ctr}} + \mathbf{p} - \mathbf{A}\mathbf{v}_i \in \mathbb{Z}_q^n,$$
      together with the local opening $\mathbf{z}_{\mathsf{ctr}} = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes \mathbf{t}_{\mathsf{ctr}}, i)$. The challenger sets $\mathsf{pk}_{\mathsf{ctr}} = \mathbf{t}_{\mathsf{ctr}}$, $\mathsf{sk}_{\mathsf{ctr}} = (i, \mathbf{r}_{\mathsf{ctr}}, \mathbf{z}_{\mathsf{ctr}})$, and $\mathsf{ask}_{\mathsf{ctr}} = \perp$, adds the mapping $\mathsf{ctr} \mapsto (i, \mathsf{pk}_{\mathsf{ctr}}, \mathsf{sk}_{\mathsf{ctr}}, \mathsf{ask}_{\mathsf{ctr}})$ to D, and defines the response to the query to be $\mathsf{pk}_{\mathsf{ctr}}$.

* **Corruption query on a counter** $c \in [\mathsf{ctr}]$**:** The challenger looks up $(i, \mathsf{pk}_c, \mathsf{sk}_c, \mathsf{ask}_c) = \mathsf{D}[c]$ and defines the response to the query to be $(\mathsf{sk}_c, \mathsf{ask}_c)$. In addition, the challenger updates $C = C \cup \{c\}$.
* **Update query on an operation** $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{Refresh}\}$ **together with a counter** $c \in [\mathsf{ctr}]$ **when** $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$**:** If $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, the challenger looks up $(i, \mathsf{pk}_c, \mathsf{sk}_c, \mathsf{ask}_c) = \mathsf{D}[c]$ and parses $\mathsf{pk}_c = \mathbf{t}_c$ and $\mathsf{sk}_c = (i, \mathbf{r}_c, \mathbf{z}_c)$. If either ($\mathsf{op} = \mathsf{Add}$ and $i \in S$) or ($\mathsf{op} = \mathsf{Remove}$ and $c \notin T$), the challenger sets the response to this query to be $\perp$. Otherwise, if $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, the challenger computes the commitment

$$\mathbf{C}_{(c)} = \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes \mathbf{t}_c) \in \mathbb{Z}_q^{n \times m}.$$

In the following, for counters $\tau, \tau' \in T \cup \{c\}$, we write $(i_\tau, \mathbf{t}_\tau)$ to denote the index and public key associated with $\mathsf{D}[\tau]$, and we write

$$\mathbf{z}_{(\tau,\tau')} = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_\tau}^\intercal \otimes \mathbf{t}_\tau, i_{\tau'})$$

to denote the respective local openings. The challenger then proceeds based on the operation:

- **Case** $\mathsf{op} = \mathsf{Add}$**:** For each $\tau \in T$, the challenger looks up $(i_\tau, \mathsf{pk}_\tau, \mathsf{sk}_\tau, \mathsf{ask}_\tau) = \mathsf{D}[\tau]$, parses $\mathsf{ask}_\tau = (i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau)$, and updates $\mathsf{ask}_\tau$ in $\mathsf{D}[\tau]$ with $(i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau + \mathbf{z}_{(c,\tau)})$. For the newly added user, the challenger computes
$$\mathsf{ask}_c = \left(i, \mathbf{r}_c, \mathbf{z}_c + \sum_{\tau \in T} \mathbf{z}_{(\tau,c)}\right)$$
and updates $\mathsf{ask}_c$ in $\mathsf{D}[c]$. Finally, the challenger updates the aggregated public key $\widetilde{\mathbf{C}} = \widetilde{\mathbf{C}} + \mathbf{C}_{(c)}$, sets $S[i] = \mathbf{t}_c$, and updates $T = T \cup \{c\}$.
- **Case** $\mathsf{op} = \mathsf{Remove}$**:** For each $\tau \in T \setminus \{c\}$, the challenger looks up $(i_\tau, \mathsf{pk}_\tau, \mathsf{sk}_\tau, \mathsf{ask}_\tau) = \mathsf{D}[\tau]$, parses $\mathsf{ask}_\tau = (i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau)$, and updates $\mathsf{ask}_\tau = (i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau - \mathbf{z}_{(c,\tau)})$ in $\mathsf{D}[\tau]$. It also sets $\mathsf{ask}_c = \perp$ in $\mathsf{D}[c]$. Finally, the challenger updates the aggregated public key $\widetilde{\mathbf{C}} = \widetilde{\mathbf{C}} - \mathbf{C}_{(c)}$, removes $i$ from $S$, and updates $T = T \setminus \{c\}$.
- **Case** $\mathsf{op} = \mathsf{Refresh}$**:** Challenger parses $\mathsf{apk} = \widetilde{\mathbf{C}}$, samples $\boldsymbol{\delta} \xleftarrow{\mathrm{R}} \{0,1\}^\lambda$ and for all $\tau \in T$, sets $\boldsymbol{\Delta}_\tau = H(\boldsymbol{\delta}, i_\tau)$. For all $\ell \in [\lambda]$, it samples,
$$\mathbf{s}_{(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n, \quad \mathbf{e}_{(\ell)} \xleftarrow{\mathrm{R}} \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\hat{m}}, \quad \mathbf{D}_{1,(\ell)} \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m} \times m}, \quad \mathbf{d}_{2,(\ell)} \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}.$$
If $\|\mathbf{e}_{(\ell)}\| \geq \sqrt{\lambda}\widetilde{\sigma}$, it resets $\mathbf{e}_{(\ell)} = 0^{\hat{m}}$. It then computes $\mathsf{ctref} = (\mathsf{ctref}_{(1)}, \ldots, \mathsf{ctref}_{(\lambda)})$, where for all $\ell \in [\lambda]$, $\mathsf{ctref}_{(\ell)} = \left(\mathbf{c}_{1,(\ell)}^\intercal, \mathbf{c}_{2,(\ell)}^\intercal, c_{3,(\ell)}\right)$ and,
$$\mathbf{c}_{1,(\ell)}^\intercal = \mathbf{s}_{(\ell)}^\intercal \mathbf{B} + \mathbf{e}_{(\ell)}^\intercal, \quad \mathbf{c}_{2,(\ell)}^\intercal = \mathbf{s}_{(\ell)}^\intercal (\mathbf{A} + \widetilde{\mathbf{C}}) + \mathbf{e}_{(\ell)}^\intercal \mathbf{D}_{1,(\ell)}, \quad c_{3,(\ell)} = \mathbf{s}_{(\ell)}^\intercal \mathbf{p} + \mathbf{e}_{(\ell)}^\intercal \mathbf{d}_{2,(\ell)} + \xi \cdot \delta_{(\ell)}.$$
Let the randomness sampled be denoted by $\rho = \left(\boldsymbol{\delta}, \left\{\mathbf{s}_{(\ell)}, \mathbf{e}_{(\ell)}, \mathbf{D}_{1,(\ell)}, \mathbf{d}_{2,(\ell)}\right\}_{\ell \in [\lambda]}\right)$. After computing ctref, for each $\tau \in T$, the challenger looks up $(i_\tau, \mathsf{pk}_\tau, \mathsf{sk}_\tau, \mathsf{ask}_\tau) = \mathsf{D}[\tau]$, parses $\mathsf{pk}_\tau = \mathbf{t}_\tau$, $\mathsf{sk}_\tau = (i_\tau, \mathbf{r}_\tau, \mathbf{z}_\tau)$, $\mathsf{ask}_\tau = (i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau)$. It computes the verification vector $\mathbf{v}_\tau = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i_\tau)$ and for all $\tau \in T, \ell \in [\lambda]$, computes,
$$\delta_{\tau,(\ell)} = \lfloor c_{3,(\ell)} + \mathbf{c}_{1,(\ell)}^\intercal (\mathbf{r}_\tau - \widetilde{\mathbf{z}}_\tau) - \mathbf{c}_{2,(\ell)}^\intercal \mathbf{v}_\tau \rceil_2.$$
If for some $\tau \in T$ and $\ell \in [\lambda]$ the value $\delta_{\tau,(\ell)}$ computed above does not match the sampled bit $\delta_{(\ell)}$, the challenger halts the experiment and outputs 0. This is the concrete instantiation of the check $S_\tau = S'$ in Definition 5.8: since each user derives the shifts $\boldsymbol{\Delta}_\tau$ from the seed it recovers from ctref, the mappings agree if and only if every user recovers $\boldsymbol{\delta}$. Otherwise, the challenger computes $\mathsf{apk}' = \sum_{\tau \in T} \mathsf{Com}\left(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_\tau}^\intercal \otimes (\mathbf{t}_\tau + \mathbf{B}\boldsymbol{\Delta}_\tau)\right)$ and sets $\widetilde{\mathbf{C}} = \mathsf{apk}'$ and $S[i_\tau] = \mathbf{t}_\tau + \mathbf{B}\boldsymbol{\Delta}_\tau$ for all $\tau \in T$.

For user $\tau \in T$, update

$$\mathsf{ask}'_\tau = \left( i_\tau, \mathbf{r}_\tau + \boldsymbol{\Delta}_\tau, \sum_{\tau' \in T} \mathsf{Open}\left(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_{\tau'}}^\top \otimes (\mathbf{t}_{\tau'} + \mathbf{B}\boldsymbol{\Delta}_{\tau'}), i_\tau\right)\right),$$

$$\mathsf{pk}'_\tau = \mathbf{t}_\tau + \mathbf{B}\boldsymbol{\Delta}_\tau, \qquad \mathsf{sk}'_\tau = \left(i_\tau, \mathbf{r}_\tau + \boldsymbol{\Delta}_\tau, \mathsf{Open}\left(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_\tau}^\top \otimes (\mathbf{t}_\tau + \mathbf{B}\boldsymbol{\Delta}_\tau), i_\tau\right)\right).$$

For all $\tau \in T$, update $\mathsf{D}[\tau] = (i_\tau, \mathsf{pk}'_\tau, \mathsf{sk}'_\tau, \mathsf{ask}'_\tau)$.

The challenger sets the response to the query to be $\left(\widetilde{\mathbf{C}}, \left\{(\tau, \mathsf{pk}_\tau, \mathsf{sk}_\tau, \mathsf{ask}_\tau)\right\}_{\tau \in T \cap C}\right)$, where $(\mathsf{pk}_\tau, \mathsf{sk}_\tau, \mathsf{ask}_\tau)$ denote the (possibly updated) keys currently stored in $\mathsf{D}[\tau]$. If the query is Refresh, the challenger also includes ctref, $\rho$ as part of the response.

After computing the responses to all of the committed pre-challenge queries, the challenger constructs the challenge ciphertext as follows:

* If $T = \emptyset$ or $T \cap C \neq \emptyset$, the challenger halts the experiment with output 0.
* Otherwise, the challenger sets $\hat{T} = T$ and $\hat{\mathbf{C}} = \widetilde{\mathbf{C}}$. It samples

$$\hat{\mathbf{s}} \xleftarrow{R} \mathbb{Z}_q^n, \qquad \hat{\mathbf{e}} \leftarrow \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\hat{m}}, \qquad \hat{\mathbf{D}}_1 \xleftarrow{R} \{0,1\}^{\hat{m} \times m}, \qquad \hat{\mathbf{d}}_2 \xleftarrow{R} \{0,1\}^{\hat{m}}.$$

If $\|\hat{\mathbf{e}}\| \geq \sqrt{\lambda}\,\widetilde{\sigma}$, it resets $\hat{\mathbf{e}} = \mathbf{0}^{\hat{m}}$. The challenger then constructs the challenge ciphertext $\mathsf{ct}^* = (\hat{\mathbf{c}}_1, \hat{\mathbf{c}}_2, \hat{c}_3)$ as follows:

$$\hat{\mathbf{c}}_1^\top = \hat{\mathbf{s}}^\top \mathbf{B} + \hat{\mathbf{e}}^\top$$
$$\hat{\mathbf{c}}_2^\top = \hat{\mathbf{s}}^\top (\mathbf{A} + \hat{\mathbf{C}}) + \hat{\mathbf{e}}^\top \hat{\mathbf{D}}_1$$
$$\hat{c}_3 = \hat{\mathbf{s}}^\top \mathbf{p} + \hat{\mathbf{e}}^\top \hat{\mathbf{d}}_2 + \xi \cdot b.$$

Finally, the challenger gives the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, the responses to each of the committed pre-challenge queries, and the challenge ciphertext $\mathsf{ct}^*$ to $\mathcal{A}$.

– **Refresh phase:** The challenger samples $\hat{\boldsymbol{\delta}} \xleftarrow{R} \{0,1\}^\lambda$ and for all $\tau \in \hat{T}$, sets $\hat{\boldsymbol{\Delta}}_\tau = H(\hat{\boldsymbol{\delta}}, i_\tau)$. For all $\ell \in [\lambda]$, it samples,

$$\hat{\mathbf{s}}_{(\ell)} \xleftarrow{R} \mathbb{Z}_q^n, \quad \hat{\mathbf{e}}_{(\ell)} \xleftarrow{R} \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\hat{m}}, \quad \hat{\mathbf{D}}_{1,(\ell)} \xleftarrow{R} \{0,1\}^{\hat{m} \times m}, \quad \hat{\mathbf{d}}_{2,(\ell)} \xleftarrow{R} \{0,1\}^{\hat{m}}.$$

If $\|\hat{\mathbf{e}}_{(\ell)}\| \geq \sqrt{\lambda}\widetilde{\sigma}$, it resets $\hat{\mathbf{e}}_{(\ell)} = 0^{\hat{m}}$. It then computes $\hat{\mathsf{ctref}} = (\hat{\mathsf{ctref}}_{(1)}, \ldots, \hat{\mathsf{ctref}}_{(\lambda)})$, where for all $\ell \in [\lambda]$, $\hat{\mathsf{ctref}}_{(\ell)} = \left(\hat{\mathbf{c}}_{1,(\ell)}^\top, \hat{\mathbf{c}}_{2,(\ell)}^\top, \hat{c}_{3,(\ell)}\right)$ and,

$$\hat{\mathbf{c}}_{1,(\ell)}^\top = \hat{\mathbf{s}}_{(\ell)}^\top \mathbf{B} + \hat{\mathbf{e}}_{(\ell)}^\top, \quad \hat{\mathbf{c}}_{2,(\ell)}^\top = \hat{\mathbf{s}}_{(\ell)}^\top (\mathbf{A} + \hat{\mathbf{C}}) + \hat{\mathbf{e}}_{(\ell)}^\top \hat{\mathbf{D}}_{1,(\ell)}, \quad \hat{c}_{3,(\ell)} = \hat{\mathbf{s}}_{(\ell)}^\top \mathbf{p} + \hat{\mathbf{e}}_{(\ell)}^\top \hat{\mathbf{d}}_{2,(\ell)} + \xi \cdot \hat{\delta}_{(\ell)}.$$

Here, we used the fact that no update query occurs between the challenge phase and the refresh phase, so the current aggregated public key $\widetilde{\mathbf{C}}$ coincides with the aggregated key $\hat{\mathbf{C}}$ from the challenge phase. After computing $\hat{\mathsf{ctref}}$, for each $\tau \in \hat{T}$, the challenger looks up $(i_\tau, \mathsf{pk}_\tau, \mathsf{sk}_\tau, \mathsf{ask}_\tau) = \mathsf{D}[\tau]$, parses $\mathsf{pk}_\tau = \hat{\mathbf{t}}_\tau$, $\mathsf{sk}_\tau = (i_\tau, \hat{\mathbf{r}}_\tau, \hat{\mathbf{z}}_\tau)$, $\mathsf{ask}_\tau = (i_\tau, \hat{\mathbf{r}}_\tau, \overline{\mathbf{z}}_\tau)$. It computes the verification vector $\mathbf{v}_\tau = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i_\tau)$ and for all $\tau \in \hat{T}, \ell \in [\lambda]$, computes,

$$\hat{\delta}_{\tau,(\ell)} = \lfloor \hat{c}_{3,(\ell)} + \hat{\mathbf{c}}_{1,(\ell)}^\top (\hat{\mathbf{r}}_\tau - \overline{\mathbf{z}}_\tau) - \hat{\mathbf{c}}_{2,(\ell)}^\top \mathbf{v}_\tau \rceil_2.$$

If for some $\tau \in \hat{T}$ and $\ell \in [\lambda]$ the value $\hat{\delta}_{\tau,(\ell)}$ computed above does not match the sampled bit $\hat{\delta}_{(\ell)}$, the challenger halts the experiment and outputs 0. Otherwise, the challenger computes $\hat{\mathsf{apk}}' = \sum_{\tau \in \hat{T}} \mathsf{Com}\left(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_\tau}^\top \otimes (\hat{\mathbf{t}}_\tau + \mathbf{B}\hat{\boldsymbol{\Delta}}_\tau)\right)$, and sets $\widetilde{\mathbf{C}} = \hat{\mathsf{apk}}'$ and $S[i_\tau] = \hat{\mathbf{t}}_\tau + \mathbf{B}\hat{\boldsymbol{\Delta}}_\tau$ for all $\tau \in \hat{T}$.

For user $\tau \in \hat{T}$, update

$$\hat{\mathsf{ask}}'_\tau = \left( i_\tau, \hat{\mathbf{r}}_\tau + \hat{\boldsymbol{\Delta}}_\tau, \sum_{\tau' \in \hat{T}} \mathsf{Open}\left(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_{\tau'}}^\top \otimes (\hat{\mathbf{t}}_{\tau'} + \mathbf{B}\hat{\boldsymbol{\Delta}}_{\tau'}), i_\tau\right)\right),$$

$$\hat{\mathsf{pk}}'_\tau = \hat{\mathbf{t}}_\tau + \mathbf{B}\hat{\boldsymbol{\Delta}}_\tau, \qquad \hat{\mathsf{sk}}'_\tau = \left(i_\tau, \hat{\mathbf{r}}_\tau + \hat{\boldsymbol{\Delta}}_\tau, \mathsf{Open}\left(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_\tau}^\top \otimes (\hat{\mathbf{t}}_\tau + \mathbf{B}\hat{\boldsymbol{\Delta}}_\tau), i_\tau\right)\right).$$

For all $\tau \in \hat{T}$, update $\mathsf{D}[\tau] = (i_\tau, \hat{\mathsf{pk}}'_\tau, \hat{\mathsf{sk}}'_\tau, \hat{\mathsf{ask}}'_\tau)$. The challenger returns $\hat{\mathsf{apk}}', \hat{\mathsf{ctref}}$ to the adversary. Note that unlike the query phase, the adversary does not see the randomness used to compute the ciphertext $\hat{\mathsf{ctref}}$ in the refresh phase.

- **Post-challenge query phase:** The adversary may now make additional (adaptive) key-generation, corruption, and update queries. The challenger computes the response to each query exactly as it processed the corresponding pre-challenge queries. The adversary is not allowed to make corruption queries on counters $c \in \hat{T}$ (i.e., on public keys that are associated with the challenge ciphertext).
- **Reveal phase:** The challenger reveals the aggregated public key $\mathsf{apk} = \widetilde{\mathsf{C}}$ and the dictionary $\mathsf{D}$ to the adversary. Namely, for all $c \le \mathsf{ctr}$, the challenger outputs $\mathsf{D}[c] = (i, \mathsf{pk}_c, \mathsf{sk}_c, \mathsf{ask}_c)$ to the adversary.
- **Output phase:** At the end of the experiment, the adversary outputs a bit $b' \in \{0,1\}$, which is the output of the experiment.

We write $\mathsf{Hyb}_0^{(b)}(\mathcal{A})$ to denote the output of an execution of $\mathsf{Hyb}_0^{(b)}$ with adversary $\mathcal{A}$ and the challenge bit $b \in \{0,1\}$.

- $\mathsf{Hyb}_1^{(b)}$: Same as $\mathsf{Hyb}_0^{(b)}$, except the challenger samples in advance the random coins used to answer all queries in the pre-challenge and challenge phases. The challenger uses these values whenever it needs to sample fresh randomness during the game. Recall that in the static security experiment, the adversary commits to all of its pre-challenge queries at the beginning of the experiment, so the setup, the full list of key-generation queries, and the full list of update queries up to the challenge phase are all fixed before setup, so all of the randomness needed by the challenger can be sampled at the beginning of the experiment. The post-challenge queries can still be adaptive (and the challenger does not sample the randomness for these queries in advance).

  Specifically, the challenger pre-samples:

  - **Setup:** $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B}) \leftarrow \mathsf{Setup}_{\mathsf{com}}(1^\lambda)$, $\mathbf{A} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n \times m}$, and $\mathbf{p} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$.
  - **Key generation:** for each key-generation query (with counter $c$), the LWE secret $\mathbf{r}_c \xleftarrow{\mathrm{R}} \{0, \ldots, \mathfrak{B}-1\}^{\hat{m}}$.
  - **Query-phase** Refresh**:** for each pre-challenge Refresh query, an independent seed $\boldsymbol{\delta}' \xleftarrow{\mathrm{R}} \{0,1\}^\lambda$ (defining $\boldsymbol{\Delta}'_i = H(\boldsymbol{\delta}', i)$) together with its encryption coins for all $\ell \in [\lambda]$, $\mathbf{s}'_{(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$, $\mathbf{e}'_{(\ell)} \leftarrow \mathcal{D}^{\hat{m}}_{\mathbb{Z},\widetilde{\sigma}}$, $\mathbf{D}'_{1,(\ell)} \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m} \times m}$, $\mathbf{d}'_{2,(\ell)} \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$.
  - **Challenge encryption:** $\hat{\mathbf{s}} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$, $\hat{\mathbf{e}} \leftarrow \mathcal{D}^{\hat{m}}_{\mathbb{Z},\widetilde{\sigma}}$, $\hat{\mathbf{D}}_1 \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m} \times m}$, $\hat{\mathbf{d}}_2 \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$.
  - **Refresh phase:** $\hat{\boldsymbol{\delta}} \xleftarrow{\mathrm{R}} \{0,1\}^\lambda$, and for all $\ell \in [\lambda]$, $\hat{\mathbf{s}}_{(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$, $\hat{\mathbf{e}}_{(\ell)} \leftarrow \mathcal{D}^{\hat{m}}_{\mathbb{Z},\widetilde{\sigma}}$, $\hat{\mathbf{D}}_{1,(\ell)} \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m} \times m}$, $\hat{\mathbf{d}}_{2,(\ell)} \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$.

  As in $\mathsf{Hyb}_0^{(b)}$, the truncation resets $\hat{\mathbf{e}} = 0^{\hat{m}}$ and, for all $\ell \in [\lambda]$, $\mathbf{e}'_{(\ell)} = 0^{\hat{m}}$ and $\hat{\mathbf{e}}_{(\ell)} = 0^{\hat{m}}$; these remain deterministic functions of the sampled coins.

- $\mathsf{Hyb}_2^{(b)}$: Same as $\mathsf{Hyb}_1^{(b)}$, except in the refresh phase the challenger samples the shift vectors of the challenge users *independently*, rather than deriving them from the random oracle. Concretely, the challenger still samples $\hat{\boldsymbol{\delta}} \xleftarrow{\mathrm{R}} \{0,1\}^\lambda$ (which is still encrypted as the message of $\hat{\mathsf{ctref}}$), but instead of setting $\hat{\boldsymbol{\Delta}}_\tau = H(\hat{\boldsymbol{\delta}}, i_\tau)$, for each $\tau \in \hat{T}$ it samples

  $$\hat{\boldsymbol{\Delta}}_\tau \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}} \quad \text{independently for each } \tau \in \hat{T}.$$

  Note that these shifts are sampled up front (which is possible since the challenge set $\hat{T}$ is determined by the committed sequence of pre-challenge queries). Additionally, the experiment immediately halts with output 0 if the value $\hat{\boldsymbol{\delta}}$ is ever queried to the random oracle $H$ as a prefix, either by the adversary at any point in the experiment, or by the challenger itself (which happens only if some refresh query samples a seed $\boldsymbol{\delta}' = \hat{\boldsymbol{\delta}}$). We refer to this event as $\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}$. The second case ensures that in $\mathsf{Hyb}_2^{(b)}$, the value $\hat{\boldsymbol{\delta}}$ is never used as an input to $H$, so replacing $H(\hat{\boldsymbol{\delta}}, \cdot)$ by a uniform random value is consistent.

- $\mathsf{Hyb}_3^{(b)}$: Same as $\mathsf{Hyb}_2^{(b)}$, except that we change how we pre-sample the secret keys of the challenge users (those $\tau \in \hat{T}$) that will be revealed during the reveal phase. We make the following changes.
  - **Key generation:** When sampling the randomness up front, for every $\tau \in \hat{T}$ the challenger samples $\mathbf{r}'_\tau \xleftarrow{R} \{0, \ldots, \mathfrak{B}\}^{\hat{m}}$ (writing $\mathbf{v}_\tau = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i_\tau)$) and sets
  $$\mathsf{pk}_\tau = \mathbf{t}_\tau = \mathbf{B}\mathbf{r}'_\tau - \mathbf{B}\hat{\boldsymbol{\Delta}}_\tau + \mathbf{p} - \mathbf{A}\mathbf{v}_\tau, \quad \mathbf{r}_\tau = \mathbf{r}'_\tau - \hat{\boldsymbol{\Delta}}_\tau, \quad \boldsymbol{\Delta}_{\mathsf{Pre},\tau} = \boldsymbol{\Delta}_{\mathsf{Post},\tau} = 0^{\hat{m}},$$
  where $\boldsymbol{\Delta}_{\mathsf{Pre},\tau}$ and $\boldsymbol{\Delta}_{\mathsf{Post},\tau}$ track the cumulative shift applied to user $\tau$'s secret before and after the challenge phase, respectively. All other users are generated as in $\mathsf{Hyb}_2^{(b)}$. Note that the challenger can carry out this step because the set $\hat{T}$ (and the corrupted set $C$) are determined by the committed sequence of pre-challenge queries, so both are known before setup.

  Note that this is the $\mathsf{Hyb}_2^{(b)}$ key generation with $\mathbf{r}_\tau$ reparametrized: since $\mathbf{B}\mathbf{r}_\tau = \mathbf{B}\mathbf{r}'_\tau - \mathbf{B}\hat{\boldsymbol{\Delta}}_\tau$, the public key still satisfies $\mathbf{t}_\tau = \mathbf{B}\mathbf{r}_\tau + \mathbf{p} - \mathbf{A}\mathbf{v}_\tau$. The only difference is the distribution of the secret: in $\mathsf{Hyb}_2^{(b)}$, the challenger samples $\mathbf{r}_\tau \xleftarrow{R} \{0, \ldots, \mathfrak{B}-1\}^{\hat{m}}$ directly, while in this experiment, the challenger samples $\mathbf{r}_\tau = \mathbf{r}'_\tau - \hat{\boldsymbol{\Delta}}_\tau$ for $\mathbf{r}'_\tau \xleftarrow{R} \{0, \ldots, \mathfrak{B}\}^{\hat{m}}$. In Claim 5.29, we show that these are statistically close.
  - During the pre-challenge query phase (resp., post-challenge query phase), whenever Refresh is run, let $\boldsymbol{\delta}'$ be the sampled seed and $\boldsymbol{\Delta}'_\tau = H(\boldsymbol{\delta}', i_\tau)$. For each $\tau \in \hat{T} \cap T$ (i.e., only for challenge users that are in the group at the time of this refresh), the challenger updates
  $$\boldsymbol{\Delta}_{\mathsf{Pre},\tau} = \boldsymbol{\Delta}_{\mathsf{Pre},\tau} + \boldsymbol{\Delta}'_\tau \qquad \big(\text{resp., } \boldsymbol{\Delta}_{\mathsf{Post},\tau} = \boldsymbol{\Delta}_{\mathsf{Post},\tau} + \boldsymbol{\Delta}'_\tau\big).$$
  - Finally, in the reveal phase, for every $\tau \in \hat{T}$ the challenger computes
  $$\mathbf{r}_\tau \coloneqq \mathbf{r}'_\tau + \boldsymbol{\Delta}_{\mathsf{Pre},\tau} + \boldsymbol{\Delta}_{\mathsf{Post},\tau}$$
  and outputs this stored $\mathbf{r}_\tau$ as the second coordinate of $\mathsf{sk}_\tau$ and $\mathsf{ask}_\tau$.

  Observe that in $\mathsf{Hyb}_3^{(b)}$ we *do not* use $\hat{\boldsymbol{\Delta}}_\tau$ when we output $\mathbf{r}_\tau$: the $-\hat{\boldsymbol{\Delta}}_\tau$ term introduced during the setup cancels out the $\hat{\boldsymbol{\Delta}}_\tau$ term introduced during the refresh. Thus, the revealed value equals the true final secret $\mathbf{r}'_\tau + \boldsymbol{\Delta}_{\mathsf{Pre},\tau} + \boldsymbol{\Delta}_{\mathsf{Post},\tau}$. For any $\tau \in \hat{T}$, the experiment uses $\hat{\boldsymbol{\Delta}}_\tau$ when computing with $\mathbf{B}\hat{\boldsymbol{\Delta}}_\tau$ (in $\mathbf{t}_\tau$), and when using the secret keys $\mathsf{sk}_\tau, \mathsf{ask}_\tau$. Below we will remove the dependence on $\mathsf{sk}_\tau, \mathsf{ask}_\tau$.

- $\mathsf{Hyb}_4^{(b)}$: Same as $\mathsf{Hyb}_3^{(b)}$, except the challenger no longer maintains the secret $\mathbf{r}_\tau$ of a challenge user $\tau \in \hat{T}$ during the experiment: it keeps the secret slot equal to $\perp$, propagates only the public-key–side quantities (the public key $\mathbf{t}_\tau$, the aggregated opening $\widetilde{\mathbf{z}}_\tau$, and the bookkeeping vectors $\boldsymbol{\Delta}_{\mathsf{Pre},\tau}, \boldsymbol{\Delta}_{\mathsf{Post},\tau}$), and reconstructs $\mathbf{r}_\tau$ only at the reveal phase. We make the following changes.
  - **Key generation:** When sampling the randomness up front, for every $\tau \in \hat{T}$ the challenger samples $\mathbf{r}'_\tau \xleftarrow{R} \{0, \ldots, \mathfrak{B}\}^{\hat{m}}$ (writing $\mathbf{v}_\tau = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i_\tau)$) and sets,
  $$\mathbf{t}_\tau = \mathbf{B}\mathbf{r}'_\tau - \mathbf{B}\hat{\boldsymbol{\Delta}}_\tau + \mathbf{p} - \mathbf{A}\mathbf{v}_\tau, \quad \mathbf{r}_\tau = \perp, \quad \boldsymbol{\Delta}_{\mathsf{Pre},\tau} = \boldsymbol{\Delta}_{\mathsf{Post},\tau} = 0^{\hat{m}}.$$
  All other users are generated as in $\mathsf{Hyb}_3^{(b)}$. Importantly, the challenger does not store the secret second coordinate of $\mathsf{sk}_\tau, \mathsf{ask}_\tau$; it stores $\mathsf{ask}_\tau = (i_\tau, \perp, \widetilde{\mathbf{z}}_\tau)$ and $\mathsf{sk}_\tau = (i_\tau, \perp, \mathbf{z}_\tau)$.
  - During the pre-challenge query phase, the challenger aborts if it receives a corruption query on any challenge user $\tau \in \hat{T}$ (consistent with $\mathsf{Hyb}_3^{(b)}$, since $\hat{T} \cap C = \varnothing$ in any non-halting execution). The remaining update queries on a challenge user $\tau \in \hat{T}$ are processed without touching $\mathbf{r}_\tau$:
    - On op $\in \{\mathsf{Add}, \mathsf{Remove}\}$ targeting counter $c$ (with index $i$): the challenger updates the aggregate and the opening as before, leaving the secret slot $\perp$,
    $$\mathsf{apk}' = \widetilde{\mathbf{C}} \pm \mathbf{C}_{(c)}, \qquad \mathsf{ask}_\tau = \big(i_\tau, \perp, \widetilde{\mathbf{z}}_\tau \pm \mathbf{z}_{(c,\tau)}\big).$$

* On op = Refresh with sampled seed $\boldsymbol{\delta}'$ and $\boldsymbol{\Delta}'_\tau = H(\boldsymbol{\delta}', i_\tau)$: the challenger computes ctref as before, but for the challenge users $\tau \in \hat{T}$ it no longer decrypts ctref to obtain $\boldsymbol{\delta}'_\tau$ and no longer checks whether $\boldsymbol{\delta}'_\tau = \boldsymbol{\delta}'$ (this check is the only place where $\mathbf{r}_\tau$ was used). It instead uses the *genuine* sampled seed $\boldsymbol{\delta}'$ and does not update $\mathbf{r}_\tau$. It updates the keys as follows:

$$\mathsf{apk}' = \sum_{\tau \in T} \mathsf{Com}\Big(\mathsf{pp}_{\mathsf{com}},\ \mathbf{u}_{i_\tau}^{\mathsf{T}} \otimes (\mathbf{t}_\tau + \mathbf{B}\boldsymbol{\Delta}'_\tau)\Big)\,.$$

For all $\tau \in \hat{T}$, it updates

$$\begin{aligned}
\mathsf{ask}'_\tau &= \left(i_\tau, \bot,\ \sum_{\tau' \in T} \mathsf{Open}\Big(\mathsf{pp}_{\mathsf{com}},\ \mathbf{u}_{i_{\tau'}}^{\mathsf{T}} \otimes (\mathbf{t}_{\tau'} + \mathbf{B}\boldsymbol{\Delta}'_{\tau'}),\ i_\tau\Big)\right)\\
\mathsf{pk}_\tau &= \mathbf{t}_\tau + \mathbf{B}\boldsymbol{\Delta}'_\tau\\
\mathsf{sk}_\tau &= \Big(i_\tau, \bot, \mathsf{Open}\Big(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_\tau}^{\mathsf{T}} \otimes (\mathbf{t}_\tau + \mathbf{B}\boldsymbol{\Delta}'_\tau),\ i_\tau\Big)\Big)\,,\\
\boldsymbol{\Delta}_{\mathsf{Pre},\tau} &= \boldsymbol{\Delta}_{\mathsf{Pre},\tau} + \boldsymbol{\Delta}'_\tau.
\end{aligned}$$

- In the refresh phase, similar to the query phase, the challenger does not use $\mathbf{r}_\tau$ for $\tau \in \hat{T}$.
- Run the post-challenge query phase like the modified pre-challenge phase, accumulating each post-challenge refresh shift into $\boldsymbol{\Delta}_{\mathsf{Post},\tau}$ instead of $\boldsymbol{\Delta}_{\mathsf{Pre},\tau}$, and never updating $\mathbf{r}_\tau$.
- In the reveal phase, for every $\tau \in \hat{T}$ the challenger reconstructs

$$\mathbf{r}_\tau = \mathbf{r}'_\tau + \boldsymbol{\Delta}_{\mathsf{Pre},\tau} + \boldsymbol{\Delta}_{\mathsf{Post},\tau}$$

and reveals the full state as in $\mathsf{Hyb}_3^{(b)}$: the aggregated public key apk together with $\mathsf{D}[c]$ for every $c \le \mathsf{ctr}$, where for $\tau \in \hat{T}$ the entries $\mathsf{ask}_\tau = (i_\tau, \mathbf{r}_\tau, \widetilde{\mathbf{z}}_\tau)$ and $\mathsf{sk}_\tau = (i_\tau, \mathbf{r}_\tau, \mathbf{z}_\tau)$ use the reconstructed $\mathbf{r}_\tau$ (the entries of all other counters are unchanged from $\mathsf{Hyb}_3^{(b)}$).

Observe that in $\mathsf{Hyb}_4^{(b)}$ the challenger *does not* use $\mathbf{r}_\tau$ for any challenge user $\tau \in \hat{T}$ at any point during the experiment. It is *only* reconstructed at reveal. Additionally, $\hat{\boldsymbol{\Delta}}_\tau$ enters the experiment *solely* through $\mathbf{B}\hat{\boldsymbol{\Delta}}_\tau$.

- $\mathsf{Hyb}_5^{(b)}$: Same as $\mathsf{Hyb}_4^{(b)}$, except wherever the experiment uses $\mathbf{B}\hat{\boldsymbol{\Delta}}_\tau$ for a challenge user $\tau \in \hat{T}$, the challenger instead uses a freshly sampled uniform vector. Concretely, for every $\tau \in \hat{T}$ the challenger samples

$$\ddot{\mathbf{t}}_\tau \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$$

up front (in place of $\hat{\boldsymbol{\Delta}}_\tau$), and substitutes $\ddot{\mathbf{t}}_\tau$ for $\mathbf{B}\hat{\boldsymbol{\Delta}}_\tau$ throughout.

- $\mathsf{Hyb}_6^{(b)}$: Same as $\mathsf{Hyb}_5^{(b)}$, except for every $\tau \in \hat{T}$ the challenger samples the public key $\mathbf{t}_\tau \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$ directly and derives the refresh-phase shift, rather than sampling $\ddot{\mathbf{t}}_\tau$ and deriving $\mathbf{t}_\tau$. We make the following change.
  - **Key generation:** When sampling the randomness up front, for every $\tau \in \hat{T}$, the challenger samples $\mathbf{r}'_\tau \xleftarrow{\mathrm{R}} \{0, \ldots, \mathfrak{B}\}^{\hat{m}}$ and $\mathbf{t}_\tau \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$ (writing $\mathbf{v}_\tau = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i_\tau)$), and sets

$$\ddot{\mathbf{t}}_\tau = \mathbf{B}\mathbf{r}'_\tau - \mathbf{t}_\tau + \mathbf{p} - \mathbf{A}\mathbf{v}_\tau, \quad \mathbf{r}_\tau = \bot, \text{ and } \boldsymbol{\Delta}_{\mathsf{Pre},\tau} = \boldsymbol{\Delta}_{\mathsf{Post},\tau} = 0^{\hat{m}}.$$

  All other users are generated as in $\mathsf{Hyb}_5^{(b)}$. The challenger continues to use $\ddot{\mathbf{t}}_\tau$ wherever $\mathsf{Hyb}_5^{(b)}$ did (in the refresh phase and the aggregated openings), now as a derived quantity.

The point of this rewriting is that $\mathbf{t}_\tau$ is now sampled *independently of* $\mathbf{A}, \mathbf{p}$. Consequently, the challenge-phase aggregate

$$\hat{\mathbf{C}} = \sum_{\tau \in \hat{T}} \mathsf{Com}\Big(\mathsf{pp}_{\mathsf{com}},\ \mathbf{u}_{i_\tau}^{\mathsf{T}} \otimes (\mathbf{t}_\tau + \mathbf{B}\boldsymbol{\Delta}_{\mathsf{Pre},\tau})\Big)$$

is a function of $\mathsf{pp}_{\mathsf{com}}$, $\{\mathbf{t}_\tau, \boldsymbol{\Delta}_{\mathsf{Pre},\tau}\}_{\tau \in \hat{T}}$, and in particular does not depend on $\mathbf{A}, \mathbf{p}$.

- $\mathsf{Hyb}_7^{(b)}$: Same as $\mathsf{Hyb}_6^{(b)}$, except in the challenge ciphertext generation the challenger no longer checks the truncation condition $\|\hat{\mathbf{e}}\| \geq \sqrt{\lambda}\,\widetilde{\sigma}$ and never overwrites $\hat{\mathbf{e}}$ with $\mathbf{0}^{\hat{m}}$. That is, the challenger uses the freshly sampled $\hat{\mathbf{e}} \leftarrow \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\hat{m}}$ as is when forming

$$\hat{\mathbf{c}}_1^\intercal = \hat{\mathbf{s}}^\intercal \mathbf{B} + \hat{\mathbf{e}}^\intercal, \quad \hat{\mathbf{c}}_2^\intercal = \hat{\mathbf{s}}^\intercal(\mathbf{A} + \hat{\mathbf{C}}) + \hat{\mathbf{e}}^\intercal \hat{\mathbf{D}}_1, \quad \hat{c}_3 = \hat{\mathbf{s}}^\intercal \mathbf{p} + \hat{\mathbf{e}}^\intercal \hat{\mathbf{d}}_2 + \xi \cdot b.$$

- $\mathsf{Hyb}_8^{(b)}$: Same as $\mathsf{Hyb}_7^{(b)}$, except the challenger programs the public parameters using the presampled challenge-encryption coins $\hat{\mathbf{D}}_1, \hat{\mathbf{d}}_2$ and the challenge-phase aggregated key $\hat{\mathbf{C}}$. We make the following changes.

  - **Setup:** For every $\tau \in \hat{T}$ the challenger samples $\mathbf{r}'_\tau \xleftarrow{\mathrm{R}} \{0, \ldots, \mathfrak{B}\}^{\hat{m}}$ and the public key $\mathbf{t}_\tau \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$ directly.

    The challenger first computes the sets $\hat{T}$ and $C$ determined by the committed sequence of queries; if $\hat{T} = \emptyset$ or $\hat{T} \cap C \neq \emptyset$, the challenger immediately halts the experiment with output 0. Otherwise, it computes

$$\hat{\mathbf{C}} = \sum_{\tau \in \hat{T}} \mathsf{Com}\Big(\mathsf{pp}_{\mathsf{com}},\ \mathbf{u}_{i_\tau}^\intercal \otimes (\mathbf{t}_\tau + \mathbf{B}\boldsymbol{\Delta}_{\mathsf{Pre},\tau})\Big),$$

    which (as observed in $\mathsf{Hyb}_6^{(b)}$) is a function of $\mathsf{pp}_{\mathsf{com}}$ and $\{\mathbf{t}_\tau, \boldsymbol{\Delta}_{\mathsf{Pre},\tau}\}_{\tau \in \hat{T}}$, and in particular does not depend on $\mathbf{A}, \mathbf{p}$. Having computed $\hat{\mathbf{C}}$, the challenger sets $\mathbf{A} = \mathbf{B}\hat{\mathbf{D}}_1 - \hat{\mathbf{C}}$ and $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_2$, where $(\hat{\mathbf{D}}_1, \hat{\mathbf{d}}_2)$ are the presampled coins of the challenge encryption.

    Finally, the shifts $\ddot{\mathbf{t}}_\tau$ above are then evaluated with these programmed $\mathbf{A}, \mathbf{p}$. Writing $\mathbf{v}_\tau = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i_\tau)$, set the refresh-phase shift as the derived quantity

$$\ddot{\mathbf{t}}_\tau = \mathbf{B}\mathbf{r}'_\tau - \mathbf{t}_\tau + \mathbf{p} - \mathbf{A}\mathbf{v}_\tau.$$

  Everything else (key generation for non-challenge users, the query phases, the refresh phase, and the reveal phase) proceeds exactly as in $\mathsf{Hyb}_7^{(b)}$, now reading the programmed $\mathbf{A}, \mathbf{p}$ from the tape.

- $\mathsf{Hyb}_9^{(b)}$: Same as $\mathsf{Hyb}_8^{(b)}$, except in the challenge ciphertext the challenger samples $\hat{\mathbf{c}}_1 \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{\hat{m}}$ uniformly, rather than setting $\hat{\mathbf{c}}_1^\intercal = \hat{\mathbf{s}}^\intercal \mathbf{B} + \hat{\mathbf{e}}^\intercal$. It then forms the remaining components of the challenge ciphertext as deterministic functions of $\hat{\mathbf{c}}_1$:

$$\Big(\hat{\mathbf{c}}_1,\ \hat{\mathbf{c}}_1^\intercal \hat{\mathbf{D}}_1,\ \hat{\mathbf{c}}_1^\intercal \hat{\mathbf{d}}_2 + \xi \cdot b\Big).$$

- $\mathsf{Hyb}_{10}^{(b)}$: Same as $\mathsf{Hyb}_9^{(b)}$, except the challenger no longer programs $\mathbf{A}, \mathbf{p}$ and instead samples them uniformly, and samples the ciphertext uniformly:

$$\mathbf{A} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n \times m}, \qquad \mathbf{p} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n, \qquad \hat{\mathbf{c}}_2 \xleftarrow{\mathrm{R}} \mathbb{Z}_q^m, \qquad \hat{c}_3 \xleftarrow{\mathrm{R}} \mathbb{Z}_q,$$

  setting $\hat{\mathsf{ct}} = (\hat{\mathbf{c}}_1,\ \hat{\mathbf{c}}_2,\ \hat{c}_3)$ with $\hat{\mathbf{c}}_1 \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{\hat{m}}$ (from $\mathsf{Hyb}_9^{(b)}$).

We write $\mathsf{Hyb}_i^{(b)}(\mathcal{A})$ to denote the output of an execution of $\mathsf{Hyb}_i^{(b)}$ with adversary $\mathcal{A}$. We now analyze each pair of adjacent distributions.

**Claim 5.16.** For all efficient adversaries $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_0^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_1^{(b)}(\mathcal{A})$ are identically distributed.

*Proof.* The two experiments are identical except for *when* the random coins are drawn: in $\mathsf{Hyb}_1^{(b)}$ the challenger samples the coins in advance (based on the committed sequence of queries), while in $\mathsf{Hyb}_0^{(b)}$, the challenger samples the coins as the queries arrive. Since the coins themselves are identically distributed, these are identical experiments, as required. □

**Claim 5.17.** Let $H$ be modeled as a random oracle. Then for all efficient adversaries $\mathcal{A}$ and all $b \in \{0,1\}$,

$$\left|\Pr[\mathsf{Hyb}_1^{(b)}(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_2^{(b)}(\mathcal{A}) = 1]\right| \;\le\; \Pr[\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}],$$

where $\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}$ is the event that $\mathcal{A}$ queries $H$ on a prefix equal to $\hat{\boldsymbol{\delta}}$.

*Proof.* The two experiments differ only in (i) how the challenge-refresh shifts $\left\{\hat{\Delta}_\tau\right\}_{\tau\in\hat{T}}$ are produced, and (ii) the added abort on $\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}$. We argue they are identically distributed conditioned on event $\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}$ not happening.

Consider an execution in which $\mathcal{A}$ never queries $H$ on a prefix equal to $\hat{\boldsymbol{\delta}}$. In $\mathsf{Hyb}_1^{(b)}$ the challenger sets $\hat{\Delta}_\tau = H(\hat{\boldsymbol{\delta}}, i_\tau)$; since the prefix $\hat{\boldsymbol{\delta}}$ is never queried by $\mathcal{A}$, the random-oracle outputs $\left\{H(\hat{\boldsymbol{\delta}}, i_\tau)\right\}_{\tau\in\hat{T}}$ are unconstrained by $\mathcal{A}$'s queries and are therefore uniform and mutually independent. This is exactly the distribution of the fresh samples $\hat{\Delta}_\tau \xleftarrow{R} \{0,1\}^{\hat{m}}$ in $\mathsf{Hyb}_2^{(b)}$. Hence, conditioned on $\neg\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}$, the two experiments are identically distributed.

The experiments proceed identically until $\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}$ occurs, so by the difference lemma the gap in output probabilities is at most $\Pr[\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}]$. □

**Claim 5.18.** Suppose $n \ge \lambda$, $m \ge 2(n+1)\log q$, $\sigma \ge \log m$, $\mathfrak{B} \in 2^{\omega(\log\lambda)}$, $q$ is prime, and suppose that the matrix commitment scheme is secure (Fact 3.5). Then for every efficient adversary $\mathcal{A}$ making at most $Q_H$ queries to the random oracle $H$, and $Q_{\mathsf{ref}}$ refresh queries,

$$\Pr_{\mathsf{Hyb}_2^{(b)}}[\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}] \;\le\; \frac{Q_H + Q_{\mathsf{ref}}}{2^\lambda} + \mathsf{negl}(\lambda),$$

where $\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}$ is the event that $\mathcal{A}$ queries $H$ on a prefix equal to $\hat{\boldsymbol{\delta}}$ during an execution of $\mathsf{Hyb}_2^{(b)}$.

*Proof.* To prove this claim, we introduce a sequence of sub-hybrid experiments that gradually remove all dependence of $\mathcal{A}$'s view on the seed $\hat{\boldsymbol{\delta}}$, and then bound $\Pr[\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}]$ in the final experiment.

**Sub-hybrid sequence for bounding $\Pr[\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}]$.** Throughout, the *challenge* ciphertext is generated honestly (with whatever $\mathbf{A}, \mathbf{p}$ the current sub-hybrid uses) and is never modified; only the refresh-phase ciphertext $\hat{\mathsf{ctref}} = (\hat{\mathsf{ctref}}_{(1)}, \ldots, \hat{\mathsf{ctref}}_{(\lambda)})$ is transformed.

- $\mathsf{G}_0, \ldots, \mathsf{G}_4$ are defined to be *identical* to $\mathsf{Hyb}_2^{(b)}, \mathsf{Hyb}_3^{(b)}, \mathsf{Hyb}_4^{(b)}, \mathsf{Hyb}_5^{(b)}, \mathsf{Hyb}_6^{(b)}$ respectively. None of the changes between $\mathsf{Hyb}_3^{(b)}$ and $\mathsf{Hyb}_6^{(b)}$ modify $\hat{\mathsf{ctref}}$; their sole purpose here is to reach a hybrid ($\mathsf{G}_4 = \mathsf{Hyb}_6^{(b)}$) in which the challenge-phase aggregate $\hat{\mathbf{C}}$ is independent of $\mathbf{A}, \mathbf{p}$, which is what makes the programming in $\mathsf{G}_5$ well-defined.
- $\mathsf{G}_5$: Same as $\mathsf{G}_4$, except in the refresh phase, for every $\ell \in [\lambda]$, the challenger no longer checks whether $\|\hat{\mathbf{e}}_{(\ell)}\| \ge \sqrt{\lambda}\,\widetilde{\sigma}$ and never overwrites $\hat{\mathbf{e}}_{(\ell)}$ with $0^{\hat{m}}$; it uses the untruncated $\hat{\mathbf{e}}_{(\ell)} \leftarrow \mathcal{D}^{\hat{m}}_{\mathbb{Z},\widetilde{\sigma}}$ as is.

Next we process the $\lambda$ coordinates of $\hat{\mathsf{ctref}}$ one at a time. For each $\ell \in [\lambda]$, we define three sub-games ($\mathsf{G}_{6,(\ell)}, \mathsf{G}_{7,(\ell)}, \mathsf{G}_{8,(\ell)}$ *for coordinate* $\ell$). At the start of cycle $\ell$, the coordinates $\ell' \in \{1, \ldots, \ell-1\}$ of $\hat{\mathsf{ctref}}$ are already uniform, and the goal of the cycle is to make $\hat{\mathsf{ctref}}_{(\ell)}$ uniform as well. After all $\lambda$ cycles, the refresh ciphertext $\hat{\mathsf{ctref}}$ is uniform and independent of $\hat{\boldsymbol{\delta}}$. Throughout, $\hat{\mathbf{C}}$ denotes the challenge-phase aggregate (which, by the $\mathsf{G}_3 \to \mathsf{G}_4$ transition, is independent of $\mathbf{A}, \mathbf{p}$). Additionally, recall the coordinate-$\ell$ refresh ciphertext

$$\hat{\mathbf{c}}^{\mathsf{T}}_{1,(\ell)} = \hat{\mathbf{s}}^{\mathsf{T}}_{(\ell)}\mathbf{B} + \hat{\mathbf{e}}^{\mathsf{T}}_{(\ell)}, \quad \hat{\mathbf{c}}^{\mathsf{T}}_{2,(\ell)} = \hat{\mathbf{s}}^{\mathsf{T}}_{(\ell)}(\mathbf{A} + \hat{\mathbf{C}}) + \hat{\mathbf{e}}^{\mathsf{T}}_{(\ell)}\hat{\mathbf{D}}_{1,(\ell)}, \quad \hat{c}_{3,(\ell)} = \hat{\mathbf{s}}^{\mathsf{T}}_{(\ell)}\mathbf{p} + \hat{\mathbf{e}}^{\mathsf{T}}_{(\ell)}\hat{\mathbf{d}}_{2,(\ell)} + \xi\cdot\hat{\delta}_{(\ell)}. \tag{5.6}$$

We now define the remaining games:

- $\mathrm{G}_{6,(\ell)}$: Same as the previous game, except the challenger first computes the sets $\hat{T}, C$ from the committed query sequence; if $\hat{T} = \varnothing$ or $\hat{T} \cap C \neq \varnothing$ it halts with output 0. Otherwise it computes $\hat{\mathbf{C}} = \sum_{\tau \in \hat{T}} \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_\tau}^{\mathsf{T}} \otimes (\mathbf{t}_\tau + \mathbf{B}\boldsymbol{\Delta}_{\mathsf{Pre},\tau}))$ (independent of $\mathbf{A}, \mathbf{p}$) and programs the public parameters using the *coordinate-$\ell$* refresh coins:

  $$\mathbf{A} \coloneqq \mathbf{B}\hat{\mathbf{D}}_{1,(\ell)} - \hat{\mathbf{C}}, \qquad \mathbf{p} \coloneqq \mathbf{B}\hat{\mathbf{d}}_{2,(\ell)}.$$

  With this programming, $\mathbf{A} + \hat{\mathbf{C}} = \mathbf{B}\hat{\mathbf{D}}_{1,(\ell)}$ and $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_{2,(\ell)}$, so the coordinate-$\ell$ components become deterministic functions of $\hat{\mathbf{c}}_{1,(\ell)}$:

  $$\hat{\mathbf{c}}_{2,(\ell)}^{\mathsf{T}} = \hat{\mathbf{c}}_{1,(\ell)}^{\mathsf{T}}\hat{\mathbf{D}}_{1,(\ell)}, \qquad \hat{c}_{3,(\ell)} = \hat{\mathbf{c}}_{1,(\ell)}^{\mathsf{T}}\hat{\mathbf{d}}_{2,(\ell)} + \xi \cdot \hat{\delta}_{(\ell)}.$$

  Everything else (i.e., the challenge ciphertext, all key generation, and the other refresh coordinates $\ell' \neq \ell$) is computed with these programmed $\mathbf{A}, \mathbf{p}$.

  Coordinates $\ell' < \ell$ have $\hat{\mathbf{c}}_{1,(\ell')}, \hat{\mathbf{c}}_{2,(\ell')}, \hat{c}_{3,(\ell')}$ already sampled as uniform and do not depend on $\mathbf{A}, \mathbf{p}$; coordinates $\ell' > \ell$ are computed honestly under the current $\mathbf{A}, \mathbf{p}$ according to Eq. (5.6).

- $\mathrm{G}_{7,(\ell)}$: Same as $\mathrm{G}_{6,(\ell)}$, except in the coordinate-$\ell$ refresh ciphertext the challenger samples

  $$\hat{\mathbf{c}}_{1,(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{\hat{m}}$$

  uniformly, rather than setting $\hat{\mathbf{c}}_{1,(\ell)}^{\mathsf{T}} = \hat{\mathbf{s}}_{(\ell)}^{\mathsf{T}}\mathbf{B} + \hat{\mathbf{e}}_{(\ell)}^{\mathsf{T}}$. It forms the remaining coordinate-$\ell$ components as the deterministic functions of $\hat{\mathbf{c}}_{1,(\ell)}$ fixed in $\mathrm{G}_6$:

  $$\hat{\mathbf{c}}_{2,(\ell)}^{\mathsf{T}} = \hat{\mathbf{c}}_{1,(\ell)}^{\mathsf{T}}\hat{\mathbf{D}}_{1,(\ell)}, \qquad \hat{c}_{3,(\ell)} = \hat{\mathbf{c}}_{1,(\ell)}^{\mathsf{T}}\hat{\mathbf{d}}_{2,(\ell)} + \xi \cdot \hat{\delta}_{(\ell)}.$$

- $\mathrm{G}_{8,(\ell)}$: Same as $\mathrm{G}_{7,(\ell)}$, except the challenger no longer programs $\mathbf{A}, \mathbf{p}$ and instead samples them uniformly, and samples the coordinate-$\ell$ ciphertext uniformly:

  $$\mathbf{A} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n \times m}, \qquad \mathbf{p} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n}, \qquad \hat{\mathbf{c}}_{2,(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{m}, \qquad \hat{c}_{3,(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q,$$

  setting $\hat{\mathrm{ctref}}_{(\ell)} = (\hat{\mathbf{c}}_{1,(\ell)},\ \hat{\mathbf{c}}_{2,(\ell)},\ \hat{c}_{3,(\ell)})$ with $\hat{\mathbf{c}}_{1,(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{\hat{m}}$ from $\mathrm{G}_{7,(\ell)}$.

**Bad-flag experiments.** For each game G in the sequence $\mathrm{G}_0, \ldots, \mathrm{G}_{8,(\lambda)}$, we write $\mathrm{G}^{\mathrm{bad}}$ for the experiment that runs G and outputs the single bit

$$\mathrm{G}^{\mathrm{bad}}(\mathcal{A}) \coloneqq \begin{cases} 1 & \text{if } H \text{ is queried on a prefix equal to } \hat{\boldsymbol{\delta}} \text{ at some point,} \\ 0 & \text{otherwise,} \end{cases}$$

*discarding* $\mathcal{A}$'s output bit. Note $\Pr[\mathrm{G}^{\mathrm{bad}}(\mathcal{A}) = 1] = \Pr_{\mathrm{G}}[\mathrm{Bad}_{\hat{\boldsymbol{\delta}}}]$ by definition, and that the flag is an efficiently computable function of the transcript together with $\hat{\boldsymbol{\delta}}$: the challenger samples $\hat{\boldsymbol{\delta}}$ itself and observes every oracle query, so it can maintain the flag online. Recall also that $\mathrm{G}_0 = \mathrm{Hyb}_2^{(b)}$ halts as soon as the flag is set (Claim 5.17), so we may assume every game halts at that point.

We now show that the probability of the bad event changes by at most a negligible amount between each adjacent pair of games in the sequence.

**Claim 5.19.** If $\mathfrak{B} \in 2^{\omega(\log \lambda)}$, and $\hat{m}$ and $|\hat{T}|$ are polynomially bounded functions of $\lambda$, then for all $b \in \{0,1\}$, $\mathrm{G}_0^{\mathrm{bad}}$ and $\mathrm{G}_1^{\mathrm{bad}}$ are statistically indistinguishable.

*Proof.* The two games differ only in how the secret of each challenge user $\tau \in \hat{T}$ is sampled and revealed. In $\mathrm{G}_0$, for every $\tau \in \hat{T}$ the challenger samples $\mathbf{r}_\tau \xleftarrow{\mathrm{R}} \{0, \ldots, \mathfrak{B}-1\}^{\hat{m}}$ and $\hat{\boldsymbol{\Delta}}_\tau \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$, sets $\mathbf{r}'_\tau = \mathbf{r}_\tau + \hat{\boldsymbol{\Delta}}_\tau$, and outputs $\mathbf{r}_\tau + \hat{\boldsymbol{\Delta}}_\tau + \boldsymbol{\Delta}_{\mathsf{Pre},\tau} + \boldsymbol{\Delta}_{\mathsf{Post},\tau}$ during the reveal phase. In $\mathrm{G}_1$, for every $\tau \in \hat{T}$ the challenger samples $\mathbf{r}'_\tau \xleftarrow{\mathrm{R}} \{0, \ldots, \mathfrak{B}\}^{\hat{m}}$ and $\hat{\boldsymbol{\Delta}}_\tau \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$, sets $\mathbf{r}_\tau = \mathbf{r}'_\tau - \hat{\boldsymbol{\Delta}}_\tau$, and outputs $\mathbf{r}'_\tau + \boldsymbol{\Delta}_{\mathsf{Pre},\tau} + \boldsymbol{\Delta}_{\mathsf{Post},\tau}$ during the reveal phase.

In both games, the rest of the view is one fixed function of $\mathbf{r}_\tau, \mathbf{r}'_\tau$, the public-key-side shift $\mathbf{B}\hat{\boldsymbol{\Delta}}_\tau$, and the bookkeeping vectors: the public key is $\mathbf{t}_\tau = \mathbf{B}\mathbf{r}'_\tau - \mathbf{B}\hat{\boldsymbol{\Delta}}_\tau + \mathbf{p} - \mathbf{A}\mathbf{v}_\tau = \mathbf{B}\mathbf{r}_\tau + \mathbf{p} - \mathbf{A}\mathbf{v}_\tau$ and evolves identically under refreshes, so it is determined by either of $\mathbf{r}_\tau, \mathbf{r}'_\tau$ together with $\mathbf{B}\hat{\boldsymbol{\Delta}}_\tau$. Thus it suffices to compare the joint distribution of the triple $(\mathbf{r}_\tau, \hat{\boldsymbol{\Delta}}_\tau, \mathbf{r}'_\tau)$ across the two games, where in $\mathrm{G}_0$ the triple is $(\mathbf{r}_\tau, \hat{\boldsymbol{\Delta}}_\tau, \mathbf{r}_\tau + \hat{\boldsymbol{\Delta}}_\tau)$ and in $\mathrm{G}_1$ it is $(\mathbf{r}'_\tau - \hat{\boldsymbol{\Delta}}_\tau, \hat{\boldsymbol{\Delta}}_\tau, \mathbf{r}'_\tau)$.

**Lemma 5.20.** For any $\mathfrak{B}, \lambda \in \mathbb{Z}$, let $\mathcal{D}_1, \mathcal{D}_2$ be the following distributions.

- **Distribution $\mathcal{D}_1$:** Sample $v \xleftarrow{R} \{0, \ldots, \mathfrak{B}-1\}$ and $u \xleftarrow{R} \{0,1\}$. Let $\mathcal{D}_1 = (v, u, u+v)$.
- **Distribution $\mathcal{D}_2$:** Sample $v' \xleftarrow{R} \{0, \ldots, \mathfrak{B}\}$ and $u \xleftarrow{R} \{0,1\}$. Let $\mathcal{D}_2 = (v'-u, u, v')$.

If $\mathfrak{B} \in 2^{\omega(\log \lambda)}$, then $\mathcal{D}_1 \approx_s \mathcal{D}_2$.

*Proof.* Since the first coordinate is the difference of the next two coordinates, it suffices to analyze the statistical distance between $(u, u+v)$ and $(u, v')$. For every $k \in \{1, \ldots, \mathfrak{B}-1\}$ and $d \in \{0,1\}$,

$$\Pr_{\mathcal{D}_1}[u+v=k, u=d] = \frac{1}{2\mathfrak{B}}.$$

At the boundary,

$$\Pr_{\mathcal{D}_1}[u+v=0, u=1] = 0, \qquad \Pr_{\mathcal{D}_1}[u+v=\mathfrak{B}, u=0] = 0,$$

while

$$\Pr_{\mathcal{D}_1}[u+v=0, u=0] = \frac{1}{2\mathfrak{B}}, \qquad \Pr_{\mathcal{D}_1}[u+v=\mathfrak{B}, u=1] = \frac{1}{2\mathfrak{B}}.$$

Under $\mathcal{D}_2$, for every $k \in \{0, \ldots, \mathfrak{B}\}$ and $d \in \{0,1\}$,

$$\Pr_{\mathcal{D}_2}[u+v=k, u=d] = \frac{1}{2(\mathfrak{B}+1)}.$$

Plugging in these values,

$$\Delta(\mathcal{D}_1, \mathcal{D}_2) = \frac{1}{2}\sum_{k,d}\left|\Pr_{\mathcal{D}_1}[u+v=k, u=d] - \Pr_{\mathcal{D}_2}[u+v=k, u=d]\right| = \frac{1}{\mathfrak{B}+1}.$$

Since $\mathfrak{B} \in 2^{\omega(\log \lambda)}$, this is negligible in $\lambda$. □

We now lift this to vectors of dimension $\hat{m}$.

**Lemma 5.21.** For any $\mathfrak{B}, \hat{m}, \lambda \in \mathbb{Z}$, let $\mathcal{D}_1, \mathcal{D}_2$ be the following distributions.

- **Distribution $\mathcal{D}_1$:** Sample $\mathbf{v} \xleftarrow{R} \{0, \ldots, \mathfrak{B}-1\}^{\hat{m}}$ and $\mathbf{u} \xleftarrow{R} \{0,1\}^{\hat{m}}$. Let $\mathcal{D}_1 = (\mathbf{v}, \mathbf{u}, \mathbf{u}+\mathbf{v})$.
- **Distribution $\mathcal{D}_2$:** Sample $\mathbf{v}' \xleftarrow{R} \{0, \ldots, \mathfrak{B}\}^{\hat{m}}$ and $\mathbf{u} \xleftarrow{R} \{0,1\}^{\hat{m}}$. Let $\mathcal{D}_2 = (\mathbf{v}'-\mathbf{u}, \mathbf{u}, \mathbf{v}')$.

If $\hat{m}$ is a polynomially bounded function of $\lambda$ and $\mathfrak{B} \in 2^{\omega(\log \lambda)}$, then $\mathcal{D}_1 \approx_s \mathcal{D}_2$.

*Proof.* A standard hybrid argument over the $\hat{m}$ coordinates, applied to Lemma 5.20, shows that the statistical distance between the distributions is at most $\hat{m}/(\mathfrak{B}+1)$. Since $\hat{m}$ is polynomial and $\mathfrak{B}$ is super-polynomial, the result follows. □

In our case the two games differ in computing the pair $(\mathbf{r}_\tau, \mathbf{r}'_\tau)$ from $\hat{\mathbf{\Delta}}_\tau$, which is precisely the substitution $(\mathbf{v}, \mathbf{u}, \mathbf{u}+\mathbf{v}) \mapsto (\mathbf{v}'-\mathbf{u}, \mathbf{u}, \mathbf{v}')$ of Lemma 5.21 (with $\mathbf{u} = \hat{\mathbf{\Delta}}_\tau$). A standard hybrid argument over the set $\hat{T}$, applied to Lemma 5.21, shows that the statistical distance between the two games is at most $\frac{\hat{m}\cdot|\hat{T}|}{\mathfrak{B}+1}$. Since both $\hat{m}$ and $|\hat{T}|$ are polynomially bounded, the claim follows. □

**Claim 5.22.** Assuming Claim 5.12 holds (conditions in Theorem 5.11 hold), for all efficient $\mathcal{A}$ and all $b \in \{0,1\}$, the views of $\mathcal{A}$ in $\mathrm{G}_1^{\mathrm{bad}}$ and $\mathrm{G}_2^{\mathrm{bad}}$ are identically distributed.

*Proof.* The games differ only in bookkeeping for the challenge users $\tau \in \hat{T}$:

- **Deferred reconstruction of $\mathbf{r}_\tau$:** In $G_2$ the secret coordinate of $\mathsf{sk}_\tau$, $\mathsf{ask}_\tau$ is stored as $\perp$ during the experiment and reconstructed as $\mathbf{r}'_\tau + \Delta_{\mathsf{Pre},\tau} + \Delta_{\mathsf{Post},\tau}$ at the reveal phase. Moreover, the secret coordinate of a challenge user is never used to form any earlier response: it appears only in the decryption check $\hat{\boldsymbol{\delta}}_\tau = \hat{\boldsymbol{\delta}}$ in the refresh phase, which $G_2$ omits. The omission does not change the response because for an honestly generated $\hat{\mathsf{ctref}}$, the computation in the proof of Claim 5.12 shows that $\hat{\boldsymbol{\delta}}_\tau$ always matches $\hat{\boldsymbol{\delta}}$. The computation applies verbatim here: the challenger maintains the keys of each $\tau \in \hat{T}$ exactly as UpdateASK does. The one difference is that after the reparametrization of $G_1$, the secret used in the check is $\mathbf{r}_\tau = \mathbf{r}'_\tau - \hat{\Delta}_\tau$ plus the accumulated query-phase shifts, whose entries lie in $\{-1, \ldots, \mathfrak{B} + Q\}$ rather than in $\{0, \ldots, \mathfrak{B} + Q\}$. Since only the bound $\|\mathbf{r}_\tau\| \le \mathfrak{B} + Q \le 2^{\lambda+1}$ is used in the proof of Claim 5.12, the same noise bound applies.
- **Early abort on corruption of a challenge user:** In $G_2$ the game aborts on a corruption query to some $\tau \in \hat{T}$. In any non-halting execution $\hat{T} \cap C = \emptyset$, so this abort coincides with the halt already present.

All other steps are unchanged, so the two games are identical. □

**Claim 5.23.** Suppose $n \ge \lambda$, $m \ge 2(n+1)\log q$, $\sigma \ge \log m$, and $q$ is prime. Then for all efficient $\mathcal{A}$ and all $b \in \{0,1\}$, the views of $\mathcal{A}$ in $G_2^{\mathsf{bad}}$ and $G_3^{\mathsf{bad}}$ are statistically indistinguishable.

*Proof.* In the entire experiment $G_2$, the shifts $\left\{\hat{\Delta}_\tau\right\}_{\tau\in\hat{T}}$ enter the experiment only through the public-key-side quantities $\left\{\mathbf{B}\hat{\Delta}_\tau\right\}_{\tau\in\hat{T}}$; in particular they no longer appear in any revealed secret. The games differ only in that $G_3$ replaces each $\mathbf{B}\hat{\Delta}_\tau$ by a fresh uniform $\ddot{\mathbf{t}}_\tau \xleftarrow{R} \mathbb{Z}_q^n$. Collecting the independent samples $\hat{\Delta}_\tau \xleftarrow{R} \{0,1\}^{\hat{m}}$ into $\hat{\mathbf{K}} = [\hat{\Delta}_\tau]_{\tau\in\hat{T}} \in \{0,1\}^{\hat{m}\times|\hat{T}|}$, which is used nowhere else, Lemma 3.4 gives

$$(\mathbf{W}, \mathbf{R},\ \mathbf{B}\hat{\mathbf{K}}) \ \approx_s\ (\mathbf{W}, \mathbf{R},\ \mathbf{U}), \qquad \mathbf{U} \xleftarrow{R} \mathbb{Z}_q^{n\times|\hat{T}|},$$

where $\mathbf{B}$ is the matrix determined by $(\mathbf{W}, \mathbf{R})$ in $\mathsf{pp}_{\mathsf{com}}$. Hence $\left\{\mathbf{B}\hat{\Delta}_\tau\right\}_\tau \approx_s \left\{\ddot{\mathbf{t}}_\tau\right\}_\tau$ jointly with the rest of the view, which is exactly the change from $G_2$ to $G_3$. □

**Claim 5.24.** For all efficient $\mathcal{A}$ and all $b \in \{0,1\}$, the views of $\mathcal{A}$ in $G_3^{\mathsf{bad}}$ and $G_4^{\mathsf{bad}}$ are identically distributed.

*Proof.* The change is only a reordering of the sampling for the challenge users. In $G_3$, for each $\tau \in \hat{T}$ the challenger samples $\ddot{\mathbf{t}}_\tau \xleftarrow{R} \mathbb{Z}_q^n$ and sets $\mathbf{t}_\tau = \mathbf{B}\mathbf{r}'_\tau - \ddot{\mathbf{t}}_\tau + \mathbf{p} - \mathbf{A}\mathbf{v}_\tau$; in $G_4$ it samples $\mathbf{t}_\tau \xleftarrow{R} \mathbb{Z}_q^n$ and sets $\ddot{\mathbf{t}}_\tau = \mathbf{B}\mathbf{r}'_\tau - \mathbf{t}_\tau + \mathbf{p} - \mathbf{A}\mathbf{v}_\tau$. For fixed $\mathbf{r}'_\tau, \mathbf{A}, \mathbf{p}, \mathbf{v}_\tau$, the map $\ddot{\mathbf{t}}_\tau \mapsto \mathbf{t}_\tau$ is a bijection on $\mathbb{Z}_q^n$, so the joint distribution of $(\mathbf{t}_\tau, \ddot{\mathbf{t}}_\tau)$ is the same under both samplings. Every other quantity is computed identically from this pair, so the games are identical. □

**Claim 5.25.** Suppose $m = \mathrm{poly}(\lambda)$. Then for all $\mathcal{A}$ and all $b \in \{0,1\}$, the views of $\mathcal{A}$ in $G_4^{\mathsf{bad}}$ and $G_5^{\mathsf{bad}}$ are statistically indistinguishable.

*Proof.* The games are identical except on the event that for some $\ell \in [\lambda]$, the sampled $\hat{\mathbf{e}}_{(\ell)} \leftarrow \mathcal{D}_{\mathbb{Z},\widetilde{\sigma}}^{\hat{m}}$ satisfies $\|\hat{\mathbf{e}}_{(\ell)}\| \ge \sqrt{\lambda}\,\widetilde{\sigma}$: in this case $G_4$ resets $\hat{\mathbf{e}}_{(\ell)} = 0^{\hat{m}}$ while $G_5$ keeps the sampled value. By the Gaussian tail bound (Lemma 3.2), each of the $\lambda \cdot \hat{m}$ sampled entries has magnitude at least $\sqrt{\lambda}\,\widetilde{\sigma}$ with probability at most $2^{-\lambda}$, so by a union bound (and since $\hat{m}$ is polynomial in $\lambda$) this event occurs with probability at most $\lambda\,\hat{m}\cdot 2^{-\lambda} = \mathrm{negl}(\lambda)$. Hence the two games are statistically close. □

Next we chain the per-coordinate cycles together. Define the boundary game $G_{8,(0)}^{\mathsf{bad}} := G_5^{\mathsf{bad}}$, so that the experiment proceeds through

$$G_5^{\mathsf{bad}} = G_{8,(0)}^{\mathsf{bad}} \ \to\ G_{6,(1)}^{\mathsf{bad}} \to G_{7,(1)}^{\mathsf{bad}} \to G_{8,(1)}^{\mathsf{bad}} \ \to\ G_{6,(2)}^{\mathsf{bad}} \to \cdots \to G_{8,(\lambda)}^{\mathsf{bad}}.$$

**Claim 5.26.** Suppose $n \ge \lambda$, $m \ge 2(n+1)\log q$, $\sigma \ge \log m$, and $q$ is prime. Then, for all adversaries $\mathcal{A}$ and all $b \in \{0,1\}$, the views of $\mathcal{A}$ in $G_{8,(\ell-1)}^{\mathsf{bad}}$ and $G_{6,(\ell)}^{\mathsf{bad}}$ are statistically indistinguishable.

*Proof.* The only difference between the two experiments is the distribution of the public parameters $(\mathbf{A}, \mathbf{p})$:

- In $\mathrm{G}^{\mathrm{bad}}_{8,(\ell-1)}$, the challenger samples $\mathbf{A} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n\times m}$ and $\mathbf{p} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n}$.
- In $\mathrm{G}^{\mathrm{bad}}_{6,(\ell)}$, the challenger sets $\mathbf{A} = \mathbf{B}\hat{\mathbf{D}}_{1,(\ell)} - \hat{\mathbf{C}}$ and $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_{2,(\ell)}$, where $\hat{\mathbf{D}}_{1,(\ell)} \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}\times m}$ and $\hat{\mathbf{d}}_{2,(\ell)} \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$ are the coordinate-$\ell$ refresh coins and $\hat{\mathbf{C}} = \sum_{\tau\in\hat{T}} \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i_\tau}^{\mathsf{T}} \otimes (\mathbf{t}_\tau + \mathbf{B}\Delta_{\mathsf{Pre},\tau}))$.

The values $(\hat{\mathbf{D}}_{1,(\ell)}, \hat{\mathbf{d}}_{2,(\ell)})$ are sampled independently of everything else in the experiments. Moreover, other than in the definition of $(\mathbf{A}, \mathbf{p})$ in $\mathrm{G}^{\mathrm{bad}}_{6,(\ell)}$, the values $(\hat{\mathbf{D}}_{1,(\ell)}, \hat{\mathbf{d}}_{2,(\ell)})$ only appear via the terms $\hat{\mathbf{e}}_{(\ell)}^{\mathsf{T}}\hat{\mathbf{D}}_{1,(\ell)}$ and $\hat{\mathbf{e}}_{(\ell)}^{\mathsf{T}}\hat{\mathbf{d}}_{2,(\ell)}$ in the coordinate-$\ell$ refresh ciphertext components

$$\hat{\mathbf{c}}_{2,(\ell)}^{\mathsf{T}} = \hat{\mathbf{s}}_{(\ell)}^{\mathsf{T}}(\mathbf{A} + \hat{\mathbf{C}}) + \hat{\mathbf{e}}_{(\ell)}^{\mathsf{T}}\hat{\mathbf{D}}_{1,(\ell)} \qquad \text{and} \qquad \hat{c}_{3,(\ell)} = \hat{\mathbf{s}}_{(\ell)}^{\mathsf{T}}\mathbf{p} + \hat{\mathbf{e}}_{(\ell)}^{\mathsf{T}}\hat{\mathbf{d}}_{2,(\ell)} + \xi \cdot \hat{\delta}_{(\ell)}.$$

In particular, the coordinates $\ell' < \ell$ of $\hat{\mathsf{ctref}}$ were made uniform in their respective $\mathrm{G}_8^{\mathrm{bad}}$ steps and no longer reference $(\hat{\mathbf{D}}_{1,(\ell)}, \hat{\mathbf{d}}_{2,(\ell)})$, the coordinates $\ell' > \ell$ use disjoint coins, and the challenge ciphertext uses its own coins $(\hat{\mathbf{D}}_1, \hat{\mathbf{d}}_2)$; so the only appearances of $(\hat{\mathbf{D}}_{1,(\ell)}, \hat{\mathbf{d}}_{2,(\ell)})$ are as above. Finally, the matrix $\hat{\mathbf{C}}$ is determined by the committed sequence of queries together with the public keys $\mathbf{t}_c \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$ for $c \in \hat{T}$. All of these values are sampled independently of $(\hat{\mathbf{D}}_{1,(\ell)}, \hat{\mathbf{d}}_{2,(\ell)})$ and of $(\mathbf{A}, \mathbf{p})$. By Lemma 3.4, the distributions

$$\left(\mathbf{W},\ \mathbf{R},\ \mathbf{B}[\hat{\mathbf{D}}_{1,(\ell)} \mid \hat{\mathbf{d}}_{2,(\ell)}],\ \hat{\mathbf{e}}_{(\ell)}^{\mathsf{T}}[\hat{\mathbf{D}}_{1,(\ell)} \mid \hat{\mathbf{d}}_{2,(\ell)}]\right)$$

and

$$\left(\mathbf{W},\ \mathbf{R},\ [\tilde{\mathbf{A}} \mid \tilde{\mathbf{p}}],\ \hat{\mathbf{e}}_{(\ell)}^{\mathsf{T}}[\hat{\mathbf{D}}_{1,(\ell)} \mid \hat{\mathbf{d}}_{2,(\ell)}]\right)$$

are statistically close, where $\tilde{\mathbf{A}} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n\times m}$ and $\tilde{\mathbf{p}} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n}$.

To conclude, we observe that the remainder of the two experiments is the *same* deterministic function of the quantities shown above along with coins that are independent of $(\hat{\mathbf{D}}_{1,(\ell)}, \hat{\mathbf{d}}_{2,(\ell)})$. Concretely, the public parameters are $\mathbf{A} = \mathbf{B}\hat{\mathbf{D}}_{1,(\ell)} - \hat{\mathbf{C}}$ and $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_{2,(\ell)}$ (respectively $\tilde{\mathbf{A}} - \hat{\mathbf{C}}$ and $\tilde{\mathbf{p}}$, which are uniform since $\hat{\mathbf{C}}$ is independent of the extractor output). Every public key of a non-challenge user, the aggregated keys, the challenge ciphertext, and every refresh coordinate $\ell' \neq \ell$ are derived from $(\mathbf{A}, \mathbf{p})$ and independent coins. The components above are computed from $\mathbf{A}$, $\mathbf{p}$, $\hat{\mathbf{s}}_{(\ell)}$ and $\hat{\mathbf{e}}_{(\ell)}^{\mathsf{T}}[\hat{\mathbf{D}}_{1,(\ell)} \mid \hat{\mathbf{d}}_{2,(\ell)}]$. Statistical closeness of the two experiments therefore follows. □

**Simulating the random oracle.** All of the reductions below simulate $H$ for $\mathcal{A}$ using a single lazily populated table $\mathcal{T}$ of query/answer pairs, shared between $\mathcal{A}$'s oracle queries and the reduction's own evaluations of $H$.

- On a fresh query $(\boldsymbol{\delta}, i) \notin \mathcal{T}$: if $\boldsymbol{\delta} = \hat{\boldsymbol{\delta}}$, set the Bad flag, halt, and output 1. Otherwise sample $\Delta \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$, add $(\boldsymbol{\delta}, i) \mapsto \Delta$ to $\mathcal{T}$, and return $\Delta$.
- On a repeated query, return the stored answer.
- Whenever the reduction itself needs a query-phase refresh shift $\Delta'_\tau = H(\boldsymbol{\delta}', i_\tau)$, where $\boldsymbol{\delta}'$ is the seed it has just sampled, it obtains $\Delta'_\tau$ by *evaluating $H$ through this same table*: it returns the stored value if $(\boldsymbol{\delta}', i_\tau) \in \mathcal{T}$, and otherwise samples $\Delta'_\tau \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$ and stores $(\boldsymbol{\delta}', i_\tau) \mapsto \Delta'_\tau$. Because the table is shared, any later query of $H(\boldsymbol{\delta}', i_\tau)$ by $\mathcal{A}$—which learns $\boldsymbol{\delta}'$ from the revealed randomness $\rho$—receives the identical value, so the simulation is consistent.

Crucially, the reduction never needs to evaluate $H$ at the prefix $\hat{\boldsymbol{\delta}}$: from $\mathrm{G}_0 = \mathrm{Hyb}_2^{(b)}$ onward the challenge-refresh shifts are sampled independently of $H$, and from $\mathrm{G}_3 = \mathrm{Hyb}_5^{(b)}$ onward they are replaced by uniform vectors $\ddot{\mathbf{t}}_\tau$ and do not appear at all. If a query-phase seed satisfies $\boldsymbol{\delta}' = \hat{\boldsymbol{\delta}}$ (which occurs with probability at most $Q_{\mathrm{ref}} \cdot 2^{-\lambda}$ over the reduction's choice of the $Q_{\mathrm{ref}}$ seeds), the reduction halts and outputs 1; this only increases its advantage.

**Claim 5.27.** Suppose the matrix commitment scheme (Fact 3.5) is secure. Then, for all efficient adversaries $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathrm{G}^{\mathrm{bad}}_{6,(\ell)}(\mathcal{A})$ and $\mathrm{G}^{\mathrm{bad}}_{7,(\ell)}(\mathcal{A})$ are computationally indistinguishable.

*Proof.* The only difference between the two experiments is the construction of coordinate $\ell$ of the refresh ciphertext: in $\mathsf{G}^{\mathsf{bad}}_{6,(\ell)}$, the challenger samples $\hat{\mathbf{s}}_{(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$ and $\hat{\mathbf{e}}_{(\ell)} \leftarrow \mathcal{D}^{\hat{m}}_{\mathbb{Z},\widetilde{\sigma}}$ and sets

$$\hat{\mathbf{c}}^{\mathsf{T}}_{1,(\ell)} = \hat{\mathbf{s}}^{\mathsf{T}}_{(\ell)}\mathbf{B} + \hat{\mathbf{e}}^{\mathsf{T}}_{(\ell)}, \qquad \hat{\mathbf{c}}^{\mathsf{T}}_{2,(\ell)} = \hat{\mathbf{s}}^{\mathsf{T}}_{(\ell)}(\mathbf{A} + \hat{\mathbf{C}}) + \hat{\mathbf{e}}^{\mathsf{T}}_{(\ell)}\hat{\mathbf{D}}_{1,(\ell)}, \qquad \hat{c}_{3,(\ell)} = \hat{\mathbf{s}}^{\mathsf{T}}_{(\ell)}\mathbf{p} + \hat{\mathbf{e}}^{\mathsf{T}}_{(\ell)}\hat{\mathbf{d}}_{2,(\ell)} + \xi \cdot \hat{\delta}_{(\ell)},$$

whereas in $\mathsf{G}^{\mathsf{bad}}_{7,(\ell)}$, the challenger samples $\hat{\mathbf{c}}_{1,(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{\hat{m}}$ and sets $\hat{\mathbf{c}}^{\mathsf{T}}_{2,(\ell)} = \hat{\mathbf{c}}^{\mathsf{T}}_{1,(\ell)}\hat{\mathbf{D}}_{1,(\ell)}$ and $\hat{c}_{3,(\ell)} = \hat{\mathbf{c}}^{\mathsf{T}}_{1,(\ell)}\hat{\mathbf{d}}_{2,(\ell)} + \xi \cdot \hat{\delta}_{(\ell)}$. Fix a bit $b \in \{0,1\}$. Suppose there exists an efficient adversary $\mathcal{A}$ where

$$\left|\Pr[\mathsf{G}^{\mathsf{bad}}_{6,(\ell)}(\mathcal{A}) = 1] - \Pr[\mathsf{G}^{\mathsf{bad}}_{7,(\ell)}(\mathcal{A}) = 1]\right| \geq \varepsilon(\lambda),$$

for some non-negligible $\varepsilon$. We use $\mathcal{A}$ to construct an adversary $\mathcal{B}$ for the matrix-commitment security game (Fact 3.5):

- **Initialization:** At the beginning of the game, algorithm $\mathcal{B}$ receives a challenge $(\mathsf{pp}_{\mathsf{com}}, \hat{\mathbf{y}})$ from the matrix-commitment challenger, where $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$ and $\hat{\mathbf{y}} \in \mathbb{Z}_q^{\hat{m}}$ is either an LWE sample $\hat{\mathbf{y}}^{\mathsf{T}} = \hat{\mathbf{s}}^{\mathsf{T}}_{(\ell)}\mathbf{B} + \hat{\mathbf{e}}^{\mathsf{T}}_{(\ell)}$ (where $\hat{\mathbf{s}}_{(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$ and $\hat{\mathbf{e}}_{(\ell)} \leftarrow \mathcal{D}^{\hat{m}}_{\mathbb{Z},\widetilde{\sigma}}$) or a uniform random vector $\hat{\mathbf{y}} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{\hat{m}}$.
- **Static commitment phase:** Algorithm $\mathcal{B}$ starts running algorithm $\mathcal{A}$, which commits to the entire sequence of pre-challenge queries it will make.
- **Setup phase:** Algorithm $\mathcal{B}$ computes the sets $\hat{T}$ and $C$ determined by the committed sequence of queries as in $\mathsf{G}^{\mathsf{bad}}_{6,(\ell)}$ and $\mathsf{G}^{\mathsf{bad}}_{7,(\ell)}$. If $\hat{T} = \varnothing$ or $\hat{T} \cap C \neq \varnothing$, algorithm $\mathcal{B}$ halts with output 0. Algorithm $\mathcal{B}$ uses the parameters $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$ from its challenge instead of sampling them itself. It samples the coordinate-$\ell$ refresh coins $\hat{\mathbf{D}}_{1,(\ell)} \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}\times m}$ and $\hat{\mathbf{d}}_{2,(\ell)} \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$, the public keys $\mathbf{t}_c \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$ for all $c \in \hat{T}$ together with the pre-challenge refresh shifts as in the earlier games, and computes

  $$\hat{\mathbf{C}} = \sum_{c\in\hat{T}} \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}^{\mathsf{T}}_{i_c} \otimes (\mathbf{t}_c + \mathbf{B}\boldsymbol{\Delta}_{\mathsf{Pre},c})),$$

  where $i_c$ denotes the index associated with counter $c$ in the committed sequence of queries. Algorithm $\mathcal{B}$ then sets $\mathbf{A} = \mathbf{B}\hat{\mathbf{D}}_{1,(\ell)} - \hat{\mathbf{C}}$ and $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_{2,(\ell)}$. Next, algorithm $\mathcal{B}$ initializes a counter $\mathsf{ctr} = 0$, a dictionary $\mathsf{D} = \varnothing$, a mapping $S = \varnothing$, a set $T = \varnothing$, the aggregated public key $\widetilde{\mathbf{C}} = \mathbf{0}^{n\times m}$, and a set of corrupted counters $C = \varnothing$, and processes the committed queries in order exactly as the challenger does in $\mathsf{G}^{\mathsf{bad}}_{6,(\ell)}$ and $\mathsf{G}^{\mathsf{bad}}_{7,(\ell)}$:

  - **Key-generation query on an index** $i \in [2^\lambda]$**:** Algorithm $\mathcal{B}$ increments the counter $\mathsf{ctr} = \mathsf{ctr}+1$. If $\mathsf{ctr} \in \hat{T}$, it sets $\mathsf{pk}_{\mathsf{ctr}} = \mathbf{t}_{\mathsf{ctr}}$ (the public key sampled at the beginning of the setup phase) and $\mathsf{sk}_{\mathsf{ctr}} = \mathsf{ask}_{\mathsf{ctr}} = \perp$. If $\mathsf{ctr} \notin \hat{T}$, it samples $\mathbf{r}_{\mathsf{ctr}} \xleftarrow{\mathrm{R}} \{0,\ldots,\mathfrak{B}-1\}^{\hat{m}}$, computes

    $$\mathbf{v}_i = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i), \qquad \mathbf{t}_{\mathsf{ctr}} = \mathbf{B}\mathbf{r}_{\mathsf{ctr}} + \mathbf{p} - \mathbf{A}\mathbf{v}_i, \qquad \mathbf{z}_{\mathsf{ctr}} = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^{\mathsf{T}} \otimes \mathbf{t}_{\mathsf{ctr}}, i),$$

    and sets $\mathsf{pk}_{\mathsf{ctr}} = \mathbf{t}_{\mathsf{ctr}}$, $\mathsf{sk}_{\mathsf{ctr}} = (i, \mathbf{r}_{\mathsf{ctr}}, \mathbf{z}_{\mathsf{ctr}})$, and $\mathsf{ask}_{\mathsf{ctr}} = \perp$. In both cases, algorithm $\mathcal{B}$ adds the mapping $\mathsf{ctr} \mapsto (i, \mathsf{pk}_{\mathsf{ctr}}, \mathsf{sk}_{\mathsf{ctr}}, \mathsf{ask}_{\mathsf{ctr}})$ to D and defines the response to the query to be $\mathsf{pk}_{\mathsf{ctr}}$.
  - **Corruption query on a counter** $c \in [\mathsf{ctr}]$**:** Algorithm $\mathcal{B}$ looks up $(i, \mathsf{pk}_c, \mathsf{sk}_c, \mathsf{ask}_c) = \mathsf{D}[c]$, defines the response to the query to be $(\mathsf{sk}_c, \mathsf{ask}_c)$, and updates $C = C \cup \{c\}$.
  - **Update query on a counter** $c \in [\mathsf{ctr}]$ **and operation** $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}, \mathsf{Refresh}\}$**:** Algorithm $\mathcal{B}$ proceeds exactly as the challenger in $\mathsf{G}^{\mathsf{bad}}_{6,(\ell)}$, $\mathsf{G}^{\mathsf{bad}}_{7,(\ell)}$ (updating $\widetilde{\mathbf{C}}$, the openings $\widetilde{\mathbf{z}}_\tau$, and the aggregated keys, leaving the secret slots of counters in $\hat{T}$ equal to $\perp$). Since Com and Open are deterministic and $\mathcal{B}$ knows all of their inputs (the public keys stored in D), it can carry out all of these updates without knowledge of $\hat{\mathbf{s}}_{(\ell)}$ or $\hat{\mathbf{e}}_{(\ell)}$ or using $\hat{\mathbf{y}}$.
- **Challenge ciphertext:** Algorithm $\mathcal{B}$ constructs the challenge ciphertext $\mathsf{ct}^* = (\hat{\mathbf{c}}_1, \hat{\mathbf{c}}_2, \hat{c}_3)$ honestly, using the freshly sampled challenge coins $(\hat{\mathbf{s}}, \hat{\mathbf{e}}, \hat{\mathbf{D}}_1, \hat{\mathbf{d}}_2)$ as in $\mathsf{G}^{\mathsf{bad}}_{6,(\ell)}$ and $\mathsf{G}^{\mathsf{bad}}_{7,(\ell)}$. (The challenge ciphertext is unaffected by this transition; only coordinate $\ell$ of the refresh ciphertext changes.)

- **Refresh phase:** Algorithm $\mathcal{B}$ constructs the refresh ciphertext $\hat{\mathsf{ctref}} = (\hat{\mathsf{ctref}}_{(1)}, \ldots, \hat{\mathsf{ctref}}_{(\lambda)})$ coordinate by coordinate:
  - for coordinates $\ell' < \ell$, it samples $\hat{\mathsf{ctref}}_{(\ell')}$ uniformly (these are already uniform in both $\mathrm{G}^{\mathrm{bad}}_{6,(\ell)}$ and $\mathrm{G}^{\mathrm{bad}}_{7,(\ell)}$);
  - for coordinate $\ell$, it sets
$$\hat{\mathbf{c}}_{1,(\ell)} = \hat{\mathbf{y}}, \qquad \hat{\mathbf{c}}^{\mathsf{T}}_{2,(\ell)} = \hat{\mathbf{y}}^{\mathsf{T}}\hat{\mathbf{D}}_{1,(\ell)}, \qquad \hat{c}_{3,(\ell)} = \hat{\mathbf{y}}^{\mathsf{T}}\hat{\mathbf{d}}_{2,(\ell)} + \xi \cdot \hat{\delta}_{(\ell)};$$
  - for coordinates $\ell' > \ell$, it samples fresh coins $(\hat{\mathbf{s}}_{(\ell')}, \hat{\mathbf{e}}_{(\ell')}, \hat{\mathbf{D}}_{1,(\ell')}, \hat{\mathbf{d}}_{2,(\ell')})$ and computes $\hat{\mathsf{ctref}}_{(\ell')}$ honestly under the current $\mathbf{A}, \mathbf{p}$ according to Eq. (5.6).

  Algorithm $\mathcal{B}$ then updates the aggregated keys and the dictionary in the refresh phase exactly as the challenger in $\mathrm{G}^{\mathrm{bad}}_{6,(\ell)}$, $\mathrm{G}^{\mathrm{bad}}_{7,(\ell)}$, and returns $(\mathsf{apk}, \hat{\mathsf{ctref}})$ to $\mathcal{A}$.
- **Post-challenge query phase:** Algorithm $\mathcal{B}$ responds to each of the adversary's adaptive key-generation, corruption, and update queries using the same procedure it used to process the committed pre-challenge queries.
- **Reveal and output phase:** Algorithm $\mathcal{B}$ performs the reveal phase exactly as in $\mathrm{G}^{\mathrm{bad}}_{6,(\ell)}$, $\mathrm{G}^{\mathrm{bad}}_{7,(\ell)}$ (reconstructing $\mathbf{r}_c$ for $c \in \hat{T}$ from $\mathbf{r}'_c, \Delta_{\mathsf{Pre},c}, \Delta_{\mathsf{Post},c}$, none of which involve $\hat{\mathbf{y}}$). Algorithm $\mathcal{B}$ outputs 1 if $\mathcal{A}$ queried $H$ on a prefix equal to $\hat{\boldsymbol{\delta}}$ at any point during the execution, and 0 otherwise. Note that $\mathcal{B}$ samples $\hat{\boldsymbol{\delta}}$ itself and observes all of $\mathcal{A}$'s oracle queries, so this condition is efficiently checkable. (We do *not* forward $\mathcal{A}$'s output bit: the purpose of this chain is to bound $\Pr[\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}]$, not the distinguishing advantage.)

Since $\mathcal{A}$ is efficient, algorithm $\mathcal{B}$ is also efficient. By construction, algorithm $\mathcal{B}$ implements the challenger in $\mathrm{G}^{\mathrm{bad}}_{6,(\ell)}$ and $\mathrm{G}^{\mathrm{bad}}_{7,(\ell)}$ exactly, except it uses its challenge $\hat{\mathbf{y}}$ to construct coordinate $\ell$ of the refresh ciphertext. Other than this coordinate, the challenger's behavior in the two experiments is identical and does not depend on $\hat{\mathbf{s}}_{(\ell)}$ or $\hat{\mathbf{e}}_{(\ell)}$. We now consider the two possibilities for $\hat{\mathbf{y}}$:

- Suppose $\hat{\mathbf{y}}^{\mathsf{T}} = \hat{\mathbf{s}}^{\mathsf{T}}_{(\ell)}\mathbf{B} + \hat{\mathbf{e}}^{\mathsf{T}}_{(\ell)}$ where $\hat{\mathbf{s}}_{(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}^n_q$ and $\hat{\mathbf{e}}_{(\ell)} \leftarrow \mathcal{D}^{\hat{m}}_{\mathbb{Z},\tilde{\sigma}}$. Since $\mathbf{A} + \hat{\mathbf{C}} = \mathbf{B}\hat{\mathbf{D}}_{1,(\ell)}$ and $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_{2,(\ell)}$, this means
$$\begin{aligned}
\hat{\mathbf{c}}^{\mathsf{T}}_{1,(\ell)} &= \hat{\mathbf{y}}^{\mathsf{T}} = \hat{\mathbf{s}}^{\mathsf{T}}_{(\ell)}\mathbf{B} + \hat{\mathbf{e}}^{\mathsf{T}}_{(\ell)} \\
\hat{\mathbf{c}}^{\mathsf{T}}_{2,(\ell)} &= \hat{\mathbf{y}}^{\mathsf{T}}\hat{\mathbf{D}}_{1,(\ell)} = \big(\hat{\mathbf{s}}^{\mathsf{T}}_{(\ell)}\mathbf{B} + \hat{\mathbf{e}}^{\mathsf{T}}_{(\ell)}\big)\hat{\mathbf{D}}_{1,(\ell)} = \hat{\mathbf{s}}^{\mathsf{T}}_{(\ell)}(\mathbf{A} + \hat{\mathbf{C}}) + \hat{\mathbf{e}}^{\mathsf{T}}_{(\ell)}\hat{\mathbf{D}}_{1,(\ell)} \\
\hat{c}_{3,(\ell)} &= \hat{\mathbf{y}}^{\mathsf{T}}\hat{\mathbf{d}}_{2,(\ell)} + \xi \cdot \hat{\delta}_{(\ell)} = \big(\hat{\mathbf{s}}^{\mathsf{T}}_{(\ell)}\mathbf{B} + \hat{\mathbf{e}}^{\mathsf{T}}_{(\ell)}\big)\hat{\mathbf{d}}_{2,(\ell)} + \xi \cdot \hat{\delta}_{(\ell)} = \hat{\mathbf{s}}^{\mathsf{T}}_{(\ell)}\mathbf{p} + \hat{\mathbf{e}}^{\mathsf{T}}_{(\ell)}\hat{\mathbf{d}}_{2,(\ell)} + \xi \cdot \hat{\delta}_{(\ell)}.
\end{aligned}$$
  Thus, in this case, algorithm $\mathcal{B}$ constructs coordinate $\ell$ of the refresh ciphertext exactly according to the specification in $\mathrm{G}^{\mathrm{bad}}_{6,(\ell)}$.
- Suppose $\hat{\mathbf{y}} \xleftarrow{\mathrm{R}} \mathbb{Z}^{\hat{m}}_q$. Then $\hat{\mathbf{c}}_{1,(\ell)} = \hat{\mathbf{y}}$ is uniform over $\mathbb{Z}^{\hat{m}}_q$, and algorithm $\mathcal{B}$ sets $\hat{\mathbf{c}}^{\mathsf{T}}_{2,(\ell)} = \hat{\mathbf{c}}^{\mathsf{T}}_{1,(\ell)}\hat{\mathbf{D}}_{1,(\ell)}$ and $\hat{c}_{3,(\ell)} = \hat{\mathbf{c}}^{\mathsf{T}}_{1,(\ell)}\hat{\mathbf{d}}_{2,(\ell)} + \xi \cdot \hat{\delta}_{(\ell)}$. This matches the distribution in $\mathrm{G}^{\mathrm{bad}}_{7,(\ell)}$.

We conclude that the distinguishing advantage of $\mathcal{B}$ in the matrix-commitment security game is at least $\varepsilon$, which contradicts the security of the matrix commitment scheme (Fact 3.5). □

**Claim 5.28.** Suppose $n \geq \lambda$, $m \geq 2(n+1)\log q$, $\sigma \geq \log m$, and $q$ is prime. Then, for all adversaries $\mathcal{A}$ and all $b \in \{0,1\}$, the views of $\mathcal{A}$ in $\mathrm{G}^{\mathrm{bad}}_{7,(\ell)}$ and $\mathrm{G}^{\mathrm{bad}}_{8,(\ell)}$ are statistically indistinguishable.

*Proof.* The only difference between the two games is the distribution of the tuple $(\mathbf{A}, \mathbf{p}, \hat{\mathbf{c}}_{2,(\ell)}, \hat{c}_{3,(\ell)})$:

- In $\mathrm{G}^{\mathrm{bad}}_{7,(\ell)}$, the challenger sets $\mathbf{A} = \mathbf{B}\hat{\mathbf{D}}_{1,(\ell)} - \hat{\mathbf{C}}$, $\mathbf{p} = \mathbf{B}\hat{\mathbf{d}}_{2,(\ell)}$, $\hat{\mathbf{c}}^{\mathsf{T}}_{2,(\ell)} = \hat{\mathbf{c}}^{\mathsf{T}}_{1,(\ell)}\hat{\mathbf{D}}_{1,(\ell)}$, and $\hat{c}_{3,(\ell)} = \hat{\mathbf{c}}^{\mathsf{T}}_{1,(\ell)}\hat{\mathbf{d}}_{2,(\ell)} + \xi \cdot \hat{\delta}_{(\ell)}$, where $\hat{\mathbf{D}}_{1,(\ell)} \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m} \times m}$ and $\hat{\mathbf{d}}_{2,(\ell)} \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}$.
- In $\mathrm{G}^{\mathrm{bad}}_{8,(\ell)}$, the challenger samples $\mathbf{A} \xleftarrow{\mathrm{R}} \mathbb{Z}^{n \times m}_q$, $\mathbf{p} \xleftarrow{\mathrm{R}} \mathbb{Z}^n_q$, $\hat{\mathbf{c}}_{2,(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}^m_q$, and $\hat{c}_{3,(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q$.

In $\mathrm{G}^{\mathrm{bad}}_{7,(\ell)}$, the challenger samples $\hat{\mathbf{c}}_{1,(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{\hat{m}}$ (which is independent of all other parameters), and the coins $\hat{\mathbf{D}}_{1,(\ell)}$ and $\hat{\mathbf{d}}_{2,(\ell)}$ are independent of each other and of all other coins. Apart from the four quantities listed above, they do not appear in the view. The claim then follows by two independent invocations of the leftover hash lemma:

- By combining Lemma 3.4 and Lemma 3.1 as in Claim 4.20, the distributions
$$\left(\mathbf{W},\, \mathbf{R},\, \hat{\mathbf{c}}_{1,(\ell)},\, \mathbf{B}\hat{\mathbf{d}}_{2,(\ell)},\, \hat{\mathbf{c}}_{1,(\ell)}^{\mathsf{T}}\hat{\mathbf{d}}_{2,(\ell)}\right) \qquad \text{and} \qquad \left(\mathbf{W},\, \mathbf{R},\, \hat{\mathbf{c}}_{1,(\ell)},\, \mathbf{p},\, \hat{c}'_3\right)$$
are statistically close, where $\mathbf{p} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$ and $\hat{c}'_3 \xleftarrow{\mathrm{R}} \mathbb{Z}_q$.
- Similarly, by combining Lemma 3.4 and Lemma 3.1, the distributions
$$\left(\mathbf{W},\, \mathbf{R},\, \hat{\mathbf{c}}_{1,(\ell)},\, \mathbf{B}\hat{\mathbf{D}}_{1,(\ell)},\, \hat{\mathbf{c}}_{1,(\ell)}^{\mathsf{T}}\hat{\mathbf{D}}_{1,(\ell)}\right) \qquad \text{and} \qquad \left(\mathbf{W},\, \mathbf{R},\, \hat{\mathbf{c}}_{1,(\ell)},\, \mathbf{U},\, \hat{\mathbf{c}}'_2\right)$$
are statistically close, where $\mathbf{U} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n\times m}$ and $\hat{\mathbf{c}}'_2 \xleftarrow{\mathrm{R}} \mathbb{Z}_q^m$.

Combining the two (the sources $\hat{\mathbf{d}}_{2,(\ell)}, \hat{\mathbf{D}}_{1,(\ell)}$ are independent, and we can invoke LHL sequentially), and using that adding $\xi \cdot \hat{\delta}_{(\ell)}$ to a uniform value and subtracting the independent $\hat{\mathbf{C}}$ from a uniform matrix preserve uniformity, the following distributions are statistically close:
$$\left(\mathbf{W},\, \mathbf{R},\, \hat{\mathbf{c}}_{1,(\ell)},\, \mathbf{B}\hat{\mathbf{D}}_{1,(\ell)} - \hat{\mathbf{C}},\, \mathbf{B}\hat{\mathbf{d}}_{2,(\ell)},\, \hat{\mathbf{c}}_{1,(\ell)}^{\mathsf{T}}\hat{\mathbf{D}}_{1,(\ell)},\, \hat{\mathbf{c}}_{1,(\ell)}^{\mathsf{T}}\hat{\mathbf{d}}_{2,(\ell)} + \xi \cdot \hat{\delta}_{(\ell)}\right)$$
and
$$\left(\mathbf{W},\, \mathbf{R},\, \hat{\mathbf{c}}_{1,(\ell)},\, \mathbf{A},\, \mathbf{p},\, \hat{\mathbf{c}}_{2,(\ell)},\, \hat{c}_{3,(\ell)}\right),$$
where $\mathbf{A} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n\times m}$, $\mathbf{p} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n$, $\hat{\mathbf{c}}_{2,(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^m$, and $\hat{c}_{3,(\ell)} \xleftarrow{\mathrm{R}} \mathbb{Z}_q$. The left distribution corresponds to $\mathrm{G}^{\mathrm{bad}}_{7,(\ell)}$ and the right to $\mathrm{G}^{\mathrm{bad}}_{8,(\ell)}$. In particular, $\hat{c}_{3,(\ell)}$ in $\mathrm{G}^{\mathrm{bad}}_{8,(\ell)}$ is independent of $\hat{\delta}_{(\ell)}$. □

Finally, in $\mathrm{G}^{\mathrm{bad}}_{8,(\lambda)}$ every coordinate $\ell \in [\lambda]$ of $\hat{\mathrm{ctref}}$ was made uniform and independent of $\hat{\delta}_{(\ell)}$ by Claim 5.28, so $\hat{\mathrm{ctref}} = (\hat{\mathrm{ctref}}_{(1)}, \ldots, \hat{\mathrm{ctref}}_{(\lambda)})$ is uniform and independent of $\hat{\boldsymbol{\delta}} = (\hat{\delta}_{(1)}, \ldots, \hat{\delta}_{(\lambda)})$.

Consequently, in $\mathrm{G}^{\mathrm{bad}}_{8,(\lambda)}$ the seed $\hat{\boldsymbol{\delta}} \xleftarrow{\mathrm{R}} \{0,1\}^\lambda$ is uniform and independent of $\mathcal{A}$'s entire view, so each of its $Q_H$ oracle queries has prefix $\hat{\boldsymbol{\delta}}$ with probability $2^{-\lambda}$. By a union bound (and accounting for the event that one of the $Q_{\mathrm{ref}}$ query-phase seeds equals $\hat{\boldsymbol{\delta}}$, which occurs with probability at most $Q_{\mathrm{ref}} \cdot 2^{-\lambda}$), $\Pr[\mathrm{Bad}_{\hat{\boldsymbol{\delta}}}] \le (Q_H + Q_{\mathrm{ref}})/2^\lambda$ in $\mathrm{G}^{\mathrm{bad}}_{8,(\lambda)}$, where $Q_H$ is the total number of oracle queries made by the adversary to the random oracle $H$.

Since the event $\mathrm{Bad}_{\hat{\boldsymbol{\delta}}}$ is an efficiently computable condition on the transcript (together with the challenger-internal value $\hat{\boldsymbol{\delta}}$), each of Claims 5.19 and 5.22 to 5.28 bounds the change in $\Pr[\mathrm{Bad}_{\hat{\boldsymbol{\delta}}}]$ across the corresponding transition. Hence, by a standard sequence-of-games argument,
$$\Pr_{\mathrm{G}_0}[\mathrm{Bad}_{\hat{\boldsymbol{\delta}}}] \;\le\; \Pr_{\mathrm{G}_{8,(\lambda)}}[\mathrm{Bad}_{\hat{\boldsymbol{\delta}}}] + \mathsf{negl}(\lambda) \;\le\; (Q_H + Q_{\mathrm{ref}}) \cdot 2^{-\lambda} + \mathsf{negl}(\lambda).$$ □

We now return to the main sequence of hybrids. The remaining transitions are related to the sub-hybrids of Claim 5.18 as follows:

- The transitions $\mathrm{Hyb}_2^{(b)} \to \mathrm{Hyb}_3^{(b)} \to \mathrm{Hyb}_4^{(b)} \to \mathrm{Hyb}_5^{(b)} \to \mathrm{Hyb}_6^{(b)}$ *are* the transitions $\mathrm{G}_0 \to \mathrm{G}_1 \to \mathrm{G}_2 \to \mathrm{G}_3 \to \mathrm{G}_4$, since $\mathrm{G}_i = \mathrm{Hyb}_{i+2}^{(b)}$ for $i \in \{0, \ldots, 4\}$. The only difference is that the G-claims compare the bad-flag experiments whereas the $\mathrm{Hyb}^{(b)}$-claims compare $\mathcal{A}$'s output; each of the arguments establishes (statistical or exact) closeness of the full transcript.
- The transitions $\mathrm{Hyb}_6^{(b)} \to \mathrm{Hyb}_7^{(b)} \to \mathrm{Hyb}_8^{(b)} \to \mathrm{Hyb}_9^{(b)} \to \mathrm{Hyb}_{10}^{(b)}$ are the *challenge-ciphertext* analogs of $\mathrm{G}_4 \to \mathrm{G}_5 \to \mathrm{G}_{6,(\ell)} \to \mathrm{G}_{7,(\ell)} \to \mathrm{G}_{8,(\ell)}$: they remove the truncation of the challenge error $\hat{\mathbf{e}}$ (rather than the refresh errors $\hat{\mathbf{e}}_{(\ell)}$), program $\mathbf{A}, \mathbf{p}$ with the challenge coins $\hat{\mathbf{D}}_1, \hat{\mathbf{d}}_2$ (rather than the coordinate-$\ell$ refresh coins), and randomize the challenge ciphertext (rather than one coordinate of $\hat{\mathrm{ctref}}$). Since there is only a single challenge ciphertext, no coordinate-by-coordinate loop is needed here.

**Claim 5.29.** If $\mathfrak{B} \in 2^{\omega(\log \lambda)}$, and $\hat{m}$ and $|\hat{T}|$ are polynomially bounded functions of $\lambda$, then for all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_2^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_3^{(b)}(\mathcal{A})$ are statistically indistinguishable.

*Proof.* Identical statement and proof to Claim 5.19. □

**Claim 5.30.** Assuming Claim 5.12 holds (conditions in Theorem 5.11 hold), for all efficient $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_3^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_4^{(b)}(\mathcal{A})$ are identically distributed.

*Proof.* Identical statement and proof to Claim 5.22. □

**Claim 5.31.** Suppose $n \geq \lambda$, $m \geq 2(n+1)\log q$, $\sigma \geq \log m$, and $q$ is prime. Then for all efficient $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_4^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_5^{(b)}(\mathcal{A})$ are statistically indistinguishable.

*Proof.* Identical statement and proof to Claim 5.23. □

**Claim 5.32.** For all efficient $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_5^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_6^{(b)}(\mathcal{A})$ are identically distributed.

*Proof.* Identical statement and proof to Claim 5.24. □

**Claim 5.33.** Suppose $m = \mathsf{poly}(\lambda)$. Then for all $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_6^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_7^{(b)}(\mathcal{A})$ are statistically indistinguishable.

*Proof.* Analogous to Claim 5.25, but applied to the challenge error rather than the refresh errors. □

**Claim 5.34.** Suppose $n \geq \lambda$, $m \geq 2(n+1)\log q$, $\sigma \geq \log m$, and $q$ is prime. Then, for all adversaries $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_7^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_8^{(b)}(\mathcal{A})$ are statistically indistinguishable.

*Proof.* The proof proceeds exactly as the proof of Claim 5.26, except that the programming uses the coins $\hat{\mathbf{D}}_1, \hat{\mathbf{d}}_2$ sampled for the challenge ciphertext in place of the refresh coins $\hat{\mathbf{D}}_{1,(\ell)}, \hat{\mathbf{d}}_{2,(\ell)}$. □

**Claim 5.35.** Suppose the matrix commitment scheme (Fact 3.5) is secure. Then, for all efficient adversaries $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_8^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_9^{(b)}(\mathcal{A})$ are computationally indistinguishable.

*Proof.* The proof proceeds very similarly to the proof of Claim 5.27 but performed for the challenge ciphertext $\hat{\mathbf{c}}_1$. We mention some concrete differences: (i) the challenge $\hat{\mathbf{y}}$ is used to form the *challenge* ciphertext, $\hat{\mathbf{c}}_1 = \hat{\mathbf{y}}$, $\hat{\mathbf{c}}_2^\intercal = \hat{\mathbf{y}}^\intercal \hat{\mathbf{D}}_1$, $\hat{c}_3 = \hat{\mathbf{y}}^\intercal \hat{\mathbf{d}}_2 + \xi \cdot b$, rather than coordinate $\ell$ of the refresh ciphertext; (ii) accordingly $\mathcal{B}$ programs $\mathbf{A} := \mathbf{B}\hat{\mathbf{D}}_1 - \hat{\mathbf{C}}$ and $\mathbf{p} := \mathbf{B}\hat{\mathbf{d}}_2$ using the *challenge* coins $(\hat{\mathbf{D}}_1, \hat{\mathbf{d}}_2)$ (this is the programming of $\mathsf{Hyb}_8^{(b)}$, justified by Claim 5.34), and constructs all $\lambda$ coordinates of $\hat{\mathsf{ct}}_{\mathsf{ref}}$ honestly under these programmed parameters using their own independent coins; and (iii) $\mathcal{B}$ outputs $\mathcal{A}$'s bit $b'$, since here the quantity being bounded is $\mathcal{A}$'s distinguishing advantage rather than $\Pr[\mathsf{Bad}_{\hat{\boldsymbol{\delta}}}]$. The random oracle is simulated as described above. The verification that $\hat{\mathbf{y}}^\intercal = \hat{\mathbf{s}}^\intercal \mathbf{B} + \hat{\mathbf{e}}^\intercal$ reproduces $\mathsf{Hyb}_8^{(b)}$ and that uniform $\hat{\mathbf{y}}$ reproduces $\mathsf{Hyb}_9^{(b)}$ is verbatim as in Claim 5.27. □

**Claim 5.36.** Suppose $n \geq \lambda$, $m \geq 2(n+1)\log q$, $\sigma \geq \log m$, and $q$ is prime. Then, for all adversaries $\mathcal{A}$ and all $b \in \{0,1\}$, the distributions $\mathsf{Hyb}_9^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_{10}^{(b)}(\mathcal{A})$ are statistically indistinguishable.

*Proof.* The proof proceeds exactly as the proof of Claim 5.28, but performed for the tuple $(\mathbf{A}, \mathbf{p}, \hat{\mathbf{c}}_2, \hat{c}_3)$ of the challenge ciphertext in place of $(\mathbf{A}, \mathbf{p}, \hat{\mathbf{c}}_{2,(\ell)}, \hat{c}_{3,(\ell)})$. □

**Completing the proof.** By definition, the challenger's behavior in $\mathsf{Hyb}_{10}^{(b)}$ is independent of the bit $b \in \{0,1\}$. Thus, $\mathsf{Hyb}_{10}^{(0)}$ and $\mathsf{Hyb}_{10}^{(1)}$ are identical experiments. Combining Claims 5.16 to 5.18 and 5.29 to 5.36, static security now follows by a standard hybrid argument. □

**Parameter instantiation.** We conclude by describing one way to instantiate the parameters of Construction 5.10 to satisfy the requirements of Theorems 5.11, 5.14 and 5.15. Take any constant $0 < \varepsilon < 1$ and choose the lattice parameters $(n, m, q, \widetilde{\sigma})$ and the width parameter $\sigma$ such that

$$n = \lambda^{1/\varepsilon} \;;\; q = 2^{\Theta(\lambda)} \;;\; m = 2(n+1)\lceil \log q \rceil \;;\; \widetilde{\sigma} = \mathsf{poly}(\lambda) \;;\; \sigma = \mathsf{poly}(\lambda) \;;\; \mathfrak{B} = 2^{\lambda}$$

where $q$ is prime, the constant in $q = 2^{\Theta(\lambda)}$ is chosen large enough so that $q/\log q > 2^{\lambda} \cdot \mathsf{poly}(\lambda)$ (for instance, the constant 2 is enough), and the $(2m^2, \sigma)$-decomposed LWE assumption holds with parameters $(n, m, q, \widetilde{\sigma})$. We now affirm that this choice of parameters satisfies our requirements:

- **Correctness:** $n, m, \sigma, \widetilde{\sigma} = \mathsf{poly}(\lambda)$, $\mathfrak{B} \leq 2^{\lambda}$, and $q/\log q > 2^{\lambda} \cdot \mathsf{poly}(\lambda)$, so Theorem 5.11 is satisfied.
- **Succinctness:** $n, m = \mathsf{poly}(\lambda)$ and $q \leq 2^{\mathsf{poly}(\lambda)}$, so Theorem 5.14 is satisfied.
- **Security:** $n \geq \lambda$, $m \geq 2(n+1)\log q$, $\sigma \geq \log m$, $q$ is prime, $\mathfrak{B} \in 2^{\omega(\log \lambda)}$, correctness holds, and the $(2m^2, \sigma)$-decomposed LWE assumption with parameters $(n, m, q, \widetilde{\sigma})$ holds. Then, the matrix commitment scheme from Fact 3.5 is secure and Theorem 5.15 is satisfied.

This instantiation relies on the polynomial hardness of the decomposed LWE assumption with a sub-exponential modulus-to-noise ratio $q/\widetilde{\sigma} = 2^{\Theta(\lambda)} = 2^{O(n^{\varepsilon})}$. We summarize this instantiation with the following corollary:

**Corollary 5.37** (Updatable DBE from Decomposed LWE)**.** Assuming the polynomial hardness of the decomposed LWE assumption (Assumption 3.3) with a sub-exponential modulus-to-noise ratio, Construction 5.10 is a statically secure updatable distributed broadcast encryption scheme in the random oracle model.

# 6 Continuous Group Key Agreement from DBE

In this section, we define continuous group key agreement (CGKA), and give two constructions: one from IncDBE which satisfies post-compromise security and is *optimally* efficient, and one from UpdDBE which satisfies both post-compromise security and forward secrecy, but at the cost of computationally linear (in the group size) forward secrecy updates.

## 6.1 Definition of Continuous Group Key Agreement

We first describe the syntax of CGKA, mostly following conventions from [ACDT20]. Users are identified with a non-cryptographic ID (for example, their phone number or email), and set up their cryptographic public parameters pp and secret state st by querying an Init algorithm (which may be defined with respect to a common reference string crs). Groups are collections of users (specified by their ID and public parameters pp) that share knowledge of an evolving secret key $K$. Each user maintains and updates their own secret state st as the group evolves via the operations Add, Remove, PCRefresh, and FSRefresh. These operations produce messages that are processed by current group members via the algorithm Process, or new members via the algorithm Join. We also maintain a dictionary $\mathbf{G}$ that consists of all pairs (ID, pp) that are currently in the group. The formal syntax is given below.

- $\mathsf{Setup}(1^{\lambda}) \to \mathsf{crs}$: The setup algorithm takes as input the security parameter $1^{\lambda}$ and outputs a common reference string crs.
- $\mathsf{Init}(\mathsf{crs}, \mathsf{ID}) \to \mathsf{pp}, \mathsf{st}$: The initialize algorithm takes as input crs and an id ID and outputs public parameters pp and secret state st.
- $\mathsf{Add}(\mathsf{st}, \mathsf{ID}, \mathsf{pp}) \to \mathsf{st}, K, m$: The add algorithm takes as input the state st of the invoking user, an id ID, and its associated public parameters pp, and outputs an updated state st, an updated shared key $K$, and a message $m$.
- $\mathsf{Remove}(\mathsf{st}, \mathsf{ID}, \mathsf{pp}) \to \mathsf{st}, K, m$: The remove algorithm takes as input the state st of the invoking user, an id ID, and its associated public parameters pp, and outputs an updated state st, an updated shared key $K$, and a message $m$.

- PCRefresh(st) $\rightarrow$ pp, st, $K$, $m$: The post-compromise refresh algorithm takes as input the state st of the invoking user and outputs updated public parameters pp, an updated state st, an updated shared key $K$, and a message $m$.
- FSRefresh(st, $\mathbf{G}$) $\rightarrow$ st, $K$, $m$, $\{\mathsf{pp}_{\mathsf{ID}}\}_{\mathsf{ID}\in\mathbf{G}}$: The forward-secrecy refresh algorithm takes as input the state st of the invoking user and the public information $\mathbf{G}$ associated with the current set of group members, and outputs an updated state st, an updated shared key $K$, a message $m$, and updated public parameters for each member of the group.
- Join(st, $m$, $\mathbf{G}$) $\rightarrow$ st, $K$: The join algorithm takes as input the state st of a user who has just been added to the group, a message $m$, and the public information $\mathbf{G}$ associated with the current set of group members. It outputs an updated state st and updated shared key $K$.
- Process(st, $m$) $\rightarrow$ st, $K$:[10] The process algorithm takes as input a state st and a message $m$ which may be the result of an Add, Remove, PCRefresh, or FSRefresh operation. It outputs an updated state st and updated shared key $K$.

**Correctness and security.** Informally, correctness requires that after every "valid" sequence of adds and removes (meaning that add operates on IDs not already in the group, while remove operates on IDs in the group) on "valid" public parameters (meaning parameters that were output by Init), all parties after running Process (or Join, in the case of a new user) on the message $m$ will obtain the same value of $K$.

Our notion of security follows the conventions of [ACDT20], although we differ slightly in that we do not allow the adversary any network control at all. In particular, we consider selective and completely passive adversaries that may leak the state st of any party at any time. However, we assume that all messages output by Add, Remove, PCRefresh, and FSRefresh are always immediately broadcast to all parties, and we do not allow the adversary to tamper with any messages or actively participate in the protocol (that is, they cannot sample their own (ID, pp) and request to be added to the group). We further discuss our choice to consider passive adversaries and provide an explicit comparison with the [ACDT20] security model at the end of this subsection.

However, we do allow the adversary to request that a user stop deleting old states, meaning its current state will include all of its previous states back until when the "no-delete" flag was first set.

We now formally define the correctness and security games for CGKA. The games are specified by a set of oracles that the adversary can call in order to drive the execution of group dynamics. The oracles call and store variables in a set of arrays defined as follows.

- **PP**: The dictionary of all (ID $\rightarrow$ public parameters) in the universe.
- **ST**: The dictionary of all (ID $\rightarrow$ secret state) in the universe.
- **G**: The dictionary of all (ID $\rightarrow$ public parameters) currently in the group.
- **GMap**: The dictionary of (epoch number $\rightarrow$ group), which records the set $\mathbf{G}$ at each epoch number $t$. This will be updated implicitly in the games below.
- **K**: The dictionary of ((epoch number, ID) $\rightarrow$ shared key).
- **CH**: The dictionary of (epoch number $\rightarrow$ flag) where flag is a boolean value indicating whether the adversary issued a challenge query during that epoch.
- **D**: The dictionary of (ID $\rightarrow$ flag), where flag is a boolean value indicating whether the adversary has requested that ID stop deleting its state.

The game begins by initializing each of the above to empty, initializing the epoch number $t = 0$, sampling crs $\leftarrow$ Setup($1^\lambda$), and sampling $b \xleftarrow{R} \{0,1\}$, which is the challenger's bit for real-or-random challenges. Then the adversary (on input crs) gets access to the following set of oracles.

- **Init**(ID)

[10] In one of our schemes, we will also have to allow Process to take $\mathbf{G}$ as input.

- Sample $(\mathsf{pp}, \mathsf{st}) \leftarrow \mathsf{Init}(\mathsf{crs}, \mathsf{ID})$.
- Add the entry $\mathbf{PP}[\mathsf{ID}] \rightarrow \mathsf{pp}$.
- Add the entry $\mathbf{ST}[\mathsf{ID}] \rightarrow \mathsf{st}$.
- Return pp.

- **Add**$(\mathsf{ID}, \mathsf{ID}')$
  - Require $\mathsf{ID} \in \mathbf{G}$, $\mathsf{ID}' \in \mathbf{PP} \setminus \mathbf{G}$.
  - $t \leftarrow t + 1$.
  - Run $(\mathsf{st}, K, m) \leftarrow \mathsf{Add}(\mathbf{ST}[\mathsf{ID}], \mathsf{ID}', \mathbf{PP}[\mathsf{ID}'])$, add the entry $\mathbf{K}[t, \mathsf{ID}] \rightarrow K$, and set $\mathbf{ST}[\mathsf{ID}] \rightarrow \mathsf{st}$.
  - Run $(\mathsf{st}, K) \leftarrow \mathsf{Join}(\mathbf{ST}[\mathsf{ID}'], m, \mathbf{G})$, add the entry $\mathbf{K}[t, \mathsf{ID}'] \rightarrow K$, and set $\mathbf{ST}[\mathsf{ID}'] = \mathsf{st}$.
  - For all $\mathsf{ID}'' \in \mathbf{G} \setminus \{\mathsf{ID}\}$, run $(\mathsf{st}, K) \leftarrow \mathsf{Process}(\mathbf{ST}[\mathsf{ID}''], m)$, add the entry $\mathbf{K}[t, \mathsf{ID}''] \rightarrow K$, and set $\mathbf{ST}[\mathsf{ID}''] \rightarrow \mathsf{st}$.
  - Add the entry $\mathbf{G}[\mathsf{ID}'] \rightarrow \mathbf{PP}[\mathsf{ID}']$.
  - Return $m$.

- **Remove**$(\mathsf{ID}, \mathsf{ID}')$
  - Require $\mathsf{ID}, \mathsf{ID}' \in \mathbf{G}$.
  - $t \leftarrow t + 1$.
  - Remove $\mathsf{ID}'$ from $\mathbf{G}$.
  - Run $(\mathsf{st}, K, m) \leftarrow \mathsf{Remove}(\mathbf{ST}[\mathsf{ID}], \mathsf{ID}', \mathbf{PP}[\mathsf{ID}'])$, add the entry $\mathbf{K}[t, \mathsf{ID}] \rightarrow K$, and set $\mathbf{ST}[\mathsf{ID}] \rightarrow \mathsf{st}$.
  - For all $\mathsf{ID}'' \in \mathbf{G} \setminus \{\mathsf{ID}\}$, run $(\mathsf{st}, K) \leftarrow \mathsf{Process}(\mathbf{ST}[\mathsf{ID}''], m)$, add the entry $\mathbf{K}[t, \mathsf{ID}''] \rightarrow K$, and set $\mathbf{ST}[\mathsf{ID}''] \rightarrow \mathsf{st}$.
  - Return $m$.

- **PCRefresh**$(\mathsf{ID})$
  - Require $\mathsf{ID} \in \mathbf{G}$.
  - $t \leftarrow t + 1$.
  - Run $(\mathsf{pp}, \mathsf{st}, K, m) \leftarrow \mathsf{PCRefresh}(\mathbf{ST}[\mathsf{ID}])$, add the entry $\mathbf{K}[t, \mathsf{ID}] \rightarrow K$, set $\mathbf{PP}[\mathsf{ID}] \rightarrow \mathsf{pp}$, $\mathbf{G}[\mathsf{ID}] \rightarrow \mathsf{pp}$, and $\mathbf{ST}[\mathsf{ID}] \rightarrow \mathsf{st}$.
  - For all $\mathsf{ID}' \in \mathbf{G} \setminus \{\mathsf{ID}\}$, run $(\mathsf{st}, K) \leftarrow \mathsf{Process}(\mathbf{ST}[\mathsf{ID}'], m)$, add the entry $\mathbf{K}[t, \mathsf{ID}'] \rightarrow K$, and set $\mathbf{ST}[\mathsf{ID}'] \rightarrow \mathsf{st}$.
  - Return $\mathsf{pp}, m$.

- **FSRefresh**$(\mathsf{ID})$
  - Require $\mathsf{ID} \in \mathbf{G}$.
  - $t \leftarrow t + 1$.
  - Run $(\mathsf{st}, K, m, \{\mathsf{pp}_{\mathsf{ID}}\}_{\mathsf{ID} \in \mathbf{G}}) \leftarrow \mathsf{FSRefresh}(\mathbf{ST}[\mathsf{ID}], \mathbf{G})$, add the entry $\mathbf{K}[t, \mathsf{ID}] \rightarrow K$, and set $\mathbf{ST}[\mathsf{ID}] \rightarrow \mathsf{st}$.
  - For each $\mathsf{ID} \in \mathbf{G}$, set $\mathbf{PP}[\mathsf{ID}] \rightarrow \mathsf{pp}_{\mathsf{ID}}$ and $\mathbf{G}[\mathsf{ID}] \rightarrow \mathsf{pp}_{\mathsf{ID}}$.
  - For all $\mathsf{ID}' \in \mathbf{G} \setminus \{\mathsf{ID}\}$, run $(\mathsf{st}, K) \leftarrow \mathsf{Process}(\mathbf{ST}[\mathsf{ID}'], m, \mathbf{G})$, add the entry $\mathbf{K}[t, \mathsf{ID}'] \rightarrow K$, and set $\mathbf{ST}[\mathsf{ID}'] \rightarrow \mathsf{st}$.
  - Return $m$.

- **NoDel**$(\mathsf{ID})$

- Require $\mathsf{ID} \in \mathbf{G}$.
- Set $\mathbf{D}[\mathsf{ID}] = 1$.[11]

- **Corrupt**($\mathsf{ID}$)
  - Require $\mathsf{ID} \in \mathbf{G}$.
  - Return $\mathbf{ST}[\mathsf{ID}]$.

- **Challenge**($t$)
  - Require $\mathbf{CH}[t] = 0$.
  - Set $\mathbf{CH}[t] = 1$.
  - Set $K_0 = \mathbf{K}[t, \mathsf{ID}]$, where $\mathsf{ID}$ is arbitrary such that there exists an entry $(t, \mathsf{ID}) \in \mathbf{K}$.
  - Set $K_1 \xleftarrow{R} \{0,1\}^\lambda$.
  - Return $K_b$.

Next, we formally define correctness.

**Definition 6.1** (CGKA Correctness)**.** CGKA is correct if for all adversaries $\mathcal{A}$ interacting in the CGKA game, it holds that for all $(t, \mathsf{ID})$, $(t, \mathsf{ID}') \in \mathbf{K}$, $\mathbf{K}[t, \mathsf{ID}] = \mathbf{K}[t, \mathsf{ID}']$.

Security of CGKA is defined with respect to some notion of what constitutes an *admissible* sequence of oracle queries. We formulate two definitions of admissible queries, one that captures only post-compromise security (PCS), and one that additionally captures forward secrecy (FS) (and is thus more permissive).

In the definitions, we make use of two functions:

- $\mathsf{q2e}(q)$ takes as input a query $q$ and outputs the value of $t$ (epoch number) at the end of the query.
- $\mathsf{e2q}(t)$ takes as input an epoch number $t$ and outputs the query $q$ that increments $t-1$ to $t$.

First, following the PCS definition given in [ACDT20], we demand that between any corrupt and challenge query, there is either a PCRefresh query, or the user is no longer in the group when the challenge query is made.

**Definition 6.2** (Admissible Queries: PCS)**.** A sequence of queries $(q_1, \ldots, q_\ell)$ in the CGKA security game with PCS is admissible if for any $i, j \in [\ell]$ such that $q_i = \textbf{Corrupt}(\mathsf{ID})$ and $q_j = \textbf{Challenge}(t)$, it holds that $\mathsf{q2e}(q_i) < t$ and either (i) there exists a $k$ with $\mathsf{q2e}(q_i) < \mathsf{q2e}(q_k) \leq t$ such that $q_k = \textbf{PCRefresh}(\mathsf{ID})$, or (ii) $\mathsf{ID} \notin \textbf{GMap}(t)$.

Next, we allow for corruption queries after challenges, as long as there is a FSRefresh directly after the challenge.

**Definition 6.3** (Admissible Queries: PCS + FS)**.** A sequence of queries $(q_1, \ldots, q_\ell)$ in the CGKA security game with PCS + FS is admissible if for any $i, j \in [\ell]$ such that $q_i = \textbf{Corrupt}(\mathsf{ID})$ and $q_j = \textbf{Challenge}(t)$:

- If $\mathsf{q2e}(q_i) < t$, either (i) there exists a $k$ with $\mathsf{q2e}(q_i) < \mathsf{q2e}(q_k) \leq t$ such that $q_k = \textbf{PCRefresh}(\mathsf{ID})$, or (ii) $\mathsf{ID} \notin \textbf{GMap}(t)$.
- $\mathsf{q2e}(q_i) \neq t$.
- If $\mathsf{q2e}(q_i) > t$, then $\mathsf{e2q}(t+1) = \textbf{FSRefresh}(\cdot)$ and there does not exist any $k \leq t$ such that $q_k = \textbf{NoDel}(\mathsf{ID})$.

Finally, security is defined as follows.

**Definition 6.4** (CGKA Security)**.** CGKA is secure with respect to a set of admissible queries (say, Definition 6.2 or Definition 6.3) if for any sequence $(q_1, \ldots, q_\ell)$ of admissible queries and any PPT adversary $\mathcal{A}$ making such queries, it holds that

$$\left| \Pr[\mathcal{A}(\mathsf{crs}) \to b] - \frac{1}{2} \right| = \mathsf{negl}(\lambda),$$

where crs is the common reference string sampled at the beginning of the game and $b$ is the real-or-random bit sampled at the beginning of the game.

[11] If this flag is set to 1, the dictionary entry $\mathbf{ST}[\mathsf{ID}]$ will *append* the new state to the old state rather than *override* the old state (whenever it is updated as a result of other oracle queries). However, we leave this implicit (as in [ACDT20]) to avoid notational clutter.

**Efficiency.** We say that a CGKA scheme without forward secrecy is *optimally efficient* if there is a fixed polynomial $p(\lambda)$ that bounds the run-time of each of Setup, Init, Add, Remove, PCRefresh, and Process, independent of the current size of the group. That is, the only operation that may depend on the size of the group is Join. This is in some sense inherent, if we consider reading the description of the group to be a part of the algorithm that users run to join the group. Formally, we consider the following definition.

**Definition 6.5** (CGKA Optimal Efficiency)**.** CGKA is optimally efficient if there is a fixed polynomial $p(\lambda)$ such that for all adversaries $\mathcal{A}$ interacting in the CGKA game, each call to Init, Add, Remove, PCRefresh, and Process runs in time at most $p(\lambda)$.

**Discussion: on passive security and comparison with [ACDT20].** As mentioned earlier, we model the adversary in our CGKA game as completely passive, meaning that it does not have the ability to craft maliciously generated messages on behalf of the participants or take any control over the network that is responsible for delivering messages. This is in the same spirit as the CGKA model of [ACDT20], which assumes honestly crafted messages and an honest delivery ordering, though our formulation is slightly more restrictive in that honest messages are immediately broadcast to all current group members.

We stress that passive CGKA security is still a demanding primitive: The adversary may learn honest users' secret states over the lifetime of the group and the security notion captures recovery from such compromises. Moreover, the black-box lower bound of [BDG+22] already holds in a passive security model. Thus, our construction gives the first plausibly practical proposal for CGKA with provable worst-case efficiency beyond what is possible for black-box PKE-based schemes.

We remark that the security of CGKA against *active* adversaries has some history of study in the "TreeKEM" paradigm (e.g., see [AJM22] and references therein), and extending the security of our DBE paradigm to active adversaries remains an important direction for future work. To begin with, we briefly discuss the original CGKA security model of [ACDT20], which does allow for some very *limited* active participation by the adversary. In particular, while the adversary cannot craft messages on behalf of the parties, it *does* have some limited control over the network that delivers the messages. At a high level, they allow the adversary to see all messages output by the participants (as we do), and in addition make the following decisions on behalf of the network: (1) Drop messages: The adversary can block the delivery of any add, remove, or refresh operation; and (2) Delay messages: The adversary can wait until a later time to deliver an add, remove, or refresh operation to any given party, but they cannot deliver two messages to the same party in the wrong order.

We expect that our protocol remains secure against this particular type of active behavior, which we briefly justify below, but we leave the formalization of this claim and extension to security against more powerful active adversaries to future work.

- First, the ability to drop messages allows the adversary to observe add, remove, and refresh operations from several parties and then pick one to send to other parties and the rest to drop. However, since add, remove, and refresh messages can be computed using *public information* in our scheme (in particular, the apk of the underlying DBE scheme), these extra messages can all be simulated by an adversary participating in our passive game.
- Next, delaying messages should not provide the adversary any more information than it has in our passive game. In particular, suppose that the adversary leaks the state of party ID at some epoch $t$. If the adversary is allowed to delay messages, then there may be some *other* parties that have not received some update messages yet and therefore may be at some epoch $t' < t$, and there may also be some *other* parties that have received some future update messages already and therefore may be at some epoch $t'' > t$. However, these discrepancies do not affect the state of party ID at epoch $t$, so the adversary will still see exactly the same state.

## 6.2 CGKA with PCS from Incremental DBE

**Theorem 6.6.** IncDBE (Definition 4.1) implies CGKA satisfying correctness (Definition 6.1), security with respect to the PCS set of admissible queries (Definitions 6.2 and 6.4), and optimal efficiency (Definition 6.5).

*Proof.* Let (IncDBE.Setup, IncDBE.KeyGen, IncDBE.IncPK, IncDBE.IncSK, IncDBE.Enc, IncDBE.Dec) be any incremental DBE for encrypting multi-bit messages (see Remark 4.8). We construct a CGKA scheme from IncDBE by sampling and broadcasting new keys $K$ after each Add, Remove, and PCRefresh. Our PCRefresh algorithm simply has the user remove itself (using its "old" public parameters pp), sample new public parameters pp′ by calling Init, and then add itself back (using its "new" public parameters pp′). The formal scheme is as follows.[12]

- $\mathsf{Setup}(1^\lambda)$: Run $\mathsf{pp} \leftarrow \mathsf{IncDBE.Setup}(1^\lambda)$ and output $\mathsf{crs} := \mathsf{pp}$.
- $\mathsf{Init}(\mathsf{crs}, \mathsf{ID})$: Sample $i \xleftarrow{R} [2^\lambda]$, run $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{IncDBE.KeyGen}(\mathsf{crs}, i)$, set $\mathsf{apk} := \bot, \mathsf{ask} := \mathsf{sk}, K := \bot$, and output
$$\mathsf{pp} := (i, \mathsf{pk}), \quad \mathsf{st} := (\mathsf{ID}, \mathsf{pp}, \mathsf{apk}, \mathsf{ask}, K).$$
- $\mathsf{Add}(\mathsf{st}, \mathsf{ID}, \mathsf{pp})$: Parse $\mathsf{st} = (\mathsf{ID}', \mathsf{pp}', \mathsf{apk}, \mathsf{ask}', K)$ and $\mathsf{pp} = (i, \mathsf{pk})$. Run the following algorithms.
  - $\mathsf{apk} \leftarrow \mathsf{IncDBE.IncPK}(\mathsf{apk}, (i, \mathsf{pk}), \mathsf{Add})$.
  - $\mathsf{ask}' \leftarrow \mathsf{IncDBE.IncSK}(\mathsf{ask}', (i, \mathsf{pk}), \mathsf{Add})$.
  - Sample $K \xleftarrow{R} \{0,1\}^\lambda$ and run $\mathsf{ct} \leftarrow \mathsf{IncDBE.Enc}(\mathsf{apk}, K)$.
  - Output
$$\mathsf{st} := (\mathsf{ID}', \mathsf{pp}', \mathsf{apk}, \mathsf{ask}', K), \quad K, \quad m := (\mathsf{Add}, i, \mathsf{pk}, \mathsf{apk}, \mathsf{ct}).$$
- $\mathsf{Remove}(\mathsf{st}, \mathsf{ID}, \mathsf{pp})$: Parse $\mathsf{st} = (\mathsf{ID}', \mathsf{pp}', \mathsf{apk}, \mathsf{ask}', K)$ and $\mathsf{pp} = (i, \mathsf{pk})$. Run the following algorithms.
  - $\mathsf{apk} \leftarrow \mathsf{IncDBE.IncPK}(\mathsf{apk}, (i, \mathsf{pk}), \mathsf{Remove})$.
  - $\mathsf{ask}' \leftarrow \mathsf{IncDBE.IncSK}(\mathsf{ask}', (i, \mathsf{pk}), \mathsf{Remove})$.
  - Sample $K \xleftarrow{R} \{0,1\}^\lambda$ and run $\mathsf{ct} \leftarrow \mathsf{IncDBE.Enc}(\mathsf{apk}, K)$.
  - Output
$$\mathsf{st}' := (\mathsf{ID}', \mathsf{pp}', \mathsf{apk}, \mathsf{ask}', K), \quad K, \quad m := (\mathsf{Remove}, i, \mathsf{pk}, \mathsf{apk}, \mathsf{ct}).$$
- $\mathsf{PCRefresh}(\mathsf{st})$: Parse $\mathsf{st} = (\mathsf{ID}, \mathsf{pp}, \mathsf{apk}, \mathsf{ask}, K)$.
  - Run $(\mathsf{st}, K, m_1) \leftarrow \mathsf{Remove}(\mathsf{st}, \mathsf{ID}, \mathsf{pp})$.
  - Sample $i' \xleftarrow{R} [2^\lambda]$, run $(\mathsf{pk}', \mathsf{sk}') \leftarrow \mathsf{IncDBE.KeyGen}(\mathsf{crs}, i')$, $\mathsf{pp}' := (i', \mathsf{pk}')$.
  - Set $\mathsf{st} := (\mathsf{ID}, \mathsf{pp}', \bot, \mathsf{sk}', \bot)$.
  - Run $(\mathsf{st}, K, m_2) \leftarrow \mathsf{Add}(\mathsf{st}, \mathsf{ID}, \mathsf{pp}')$.
  - Output
$$\mathsf{pp}', \quad \mathsf{st}, \quad K, \quad m := (m_1, m_2).$$
- $\mathsf{Join}(\mathsf{st}, m, \mathbf{G})$: Parse $\mathsf{st} = (\mathsf{ID}, \mathsf{pp}, \bot, \mathsf{ask}, \bot)$ and $m = (\mathsf{Add}, i, \mathsf{pk}, \mathsf{apk}, \mathsf{ct})$.
  - For $(\mathsf{ID}_j, (j, \mathsf{pk}_j)) \in \mathbf{G}$, run $\mathsf{ask} \leftarrow \mathsf{IncDBE.IncSK}(\mathsf{ask}, (j, \mathsf{pk}_j), \mathsf{Add})$.
  - Output
$$\mathsf{st} := (\mathsf{ID}, \mathsf{pp}, \mathsf{apk}, \mathsf{ask}, K), \quad K.$$
- $\mathsf{Process}(\mathsf{st}, m)$: Parse $\mathsf{st} = (\mathsf{ID}, \mathsf{pp}, \mathsf{apk}, \mathsf{ask}, K)$. We have three cases, depending on whether $m$ is the result of an Add, Remove, or PCRefresh.
  - Add:
    - Parse $m = (\mathsf{Add}, i, \mathsf{pk}, \mathsf{apk}', \mathsf{ct})$.
    - $\mathsf{ask} \leftarrow \mathsf{IncDBE.IncSK}(\mathsf{ask}, (i, \mathsf{pk}), \mathsf{Add})$.
    - $K \leftarrow \mathsf{IncDBE.Dec}(\mathsf{ask}, \mathsf{ct})$.

[12] While all IncDBE algorithms additionally take the common reference string crs as input, we suppress this to avoid notational clutter.

* Output

$$\mathsf{st} \coloneqq (\mathsf{ID}, \mathsf{pp}, \mathsf{apk}', \mathsf{ask}, K), \quad K.$$

– Remove:

* Parse $m = (\mathsf{Remove}, i, \mathsf{pk}, \mathsf{apk}', \mathsf{ct})$.
* $\mathsf{ask} \leftarrow \mathsf{IncDBE.IncSK}(\mathsf{ask}, (i, \mathsf{pk}), \mathsf{Remove})$.
* $K \leftarrow \mathsf{IncDBE.Dec}(\mathsf{ask}, \mathsf{ct})$.
* Output

$$\mathsf{st} \coloneqq (\mathsf{ID}, \mathsf{pp}, \mathsf{apk}', \mathsf{ask}, K), \quad K.$$

– PCRefresh:

* Parse $m = ((\mathsf{Remove}, i, \mathsf{pk}, \mathsf{apk}_1, \mathsf{ct}_1), (\mathsf{Add}, i', \mathsf{pk}', \mathsf{apk}_2, \mathsf{ct}_2))$.
* $\mathsf{ask} \leftarrow \mathsf{IncDBE.IncSK}(\mathsf{ask}, (i, \mathsf{pk}), \mathsf{Remove})$.
* $\mathsf{ask} \leftarrow \mathsf{IncDBE.IncSK}(\mathsf{ask}, (i', \mathsf{pk}'), \mathsf{Add})$.
* $K \leftarrow \mathsf{IncDBE.Dec}(\mathsf{ask}, \mathsf{ct}_2)$.
* Output

$$\mathsf{st} \coloneqq (\mathsf{ID}, \mathsf{pp}, \mathsf{apk}_2, \mathsf{ask}, K), \quad K.$$

Now, we show that given any sequence $(q_1, \ldots, q_\ell)$ of admissible queries according to Definition 6.2 and any PPT adversary $\mathcal{A}$ such that

$$\left| \Pr[\mathcal{A}(\mathsf{crs}) \to b] - \frac{1}{2} \right| = \mathsf{non\text{-}negl}(\lambda),$$

there exists an adversary $\mathcal{B}$ with $\mathsf{non\text{-}negl}(\lambda)$ advantage in the Incremental DBE security game (Definition 4.6). By a standard hybrid argument, it suffices to consider sequences $(q_1, \ldots, q_\ell)$ with exactly one **Challenge** query. Let $t$ be the epoch number that is input to the **Challenge** query, and let $q^* = \mathsf{e2q}(t)$ be the query to **Add**, **Remove**, or **PCRefresh** that increments $t-1$ to $t$

We define $\mathcal{B}$ as follows. It maintains an (initially empty) map **Keys**: $j \to (i, \mathsf{pk}, \mathsf{sk})$, an (initially empty) map **ID**: $\mathsf{ID} \to (i, \mathsf{pk}, \mathsf{ask})$, an (initially empty) set **G** of ID, and variables **APK** and **K**. We will sometimes refer to the three values of **Keys**$[j]$ and **ID**$[\mathsf{ID}]$ as **Keys**$[j]_1$, **Keys**$[j]_2$, **Keys**$[j]_3$, and **ID**$[\mathsf{ID}]_1$, **ID**$[\mathsf{ID}]_2$, **ID**$[\mathsf{ID}]_3$, respectively. Let $T$ be the total number of **Init** and **PCRefresh** queries among $(q_1, \ldots, q_\ell)$, equivalently the number of times that the Init algorithm is run. Let $T^* \subseteq [T]$ be the subset of $j \in [T]$ such that the $j$'th call to Init results in public parameters pp such that there exists a **Corrupt**(ID) query where pp is associated with ID at the time of this query. This can be determined by inspecting the sequence $(q_1, \ldots, q_\ell)$. $\mathcal{B}$ begins by doing the following.

For $j \in [T]$:

- Sample $i \xleftarrow{R} [2^\lambda]$, query **KeyGen**$(i)$ and receive pk.
- If $j \in T^*$, query **Corrupt**$(i)$, receive sk, and add $j \to (i, \mathsf{pk}, \mathsf{sk})$ to **Keys**.
- Otherwise, add $j \to (i, \mathsf{pk}, \perp)$ to **Keys**.

Next, $\mathcal{B}$ will answer $\mathcal{A}$'s queries as follows, setting counter $j = 1$.

For $q \in [q_1, \ldots, q_\ell]$:

- If $q =$ **Init**(ID):
  - Parse **Keys**$[j]$ as $(i, \mathsf{pk}, \mathsf{sk})$. Add $\mathsf{ID} \to (i, \mathsf{pk}, \mathsf{sk})$ to **ID**.
  - Return $(i, \mathsf{pk})$.
  - Increment $j \leftarrow j + 1$.

- If $q = \mathbf{Add}(\mathsf{ID}, \mathsf{ID}')$:
  - Add $\mathsf{ID}'$ to $\mathbf{G}$. Parse $\mathbf{ID}[\mathsf{ID}'] = (i', \mathsf{pk}', \mathsf{sk}')$. Query $\mathbf{Update}(i', \mathsf{Add})$ and receive apk.
  - For all $\mathsf{ID}'' \in \mathbf{G}$ such that $\mathbf{ID}[\mathsf{ID}'']_3 \neq \bot$:
    - Let $\mathsf{ask} := \mathbf{ID}[\mathsf{ID}'']_3$.
    - $\mathsf{ask} \leftarrow \mathsf{IncDBE.IncSK}(\mathsf{ask}, (i', \mathsf{pk}'), \mathsf{Add})$.
    - Set $\mathbf{ID}[\mathsf{ID}'']_3 := \mathsf{ask}$.
  - If $q \neq q^*$, sample $K \xleftarrow{R} \{0,1\}^\lambda$ and run $\mathsf{ct} \leftarrow \mathsf{IncDBE.Enc}(\mathsf{apk}, K)$.
  - If $q = q^*$, sample $K, K' \xleftarrow{R} \{0,1\}^\lambda$, query $\mathbf{Challenge}(K, K')$ and receive ct.
  - Set $\mathbf{APK} := \mathsf{apk}, \mathbf{K} := K$, and return $m := (\mathsf{Add}, i', \mathsf{pk}', \mathsf{apk}, \mathsf{ct})$.
- If $q = \mathbf{Remove}(\mathsf{ID}, \mathsf{ID}')$:
  - Remove $\mathsf{ID}'$ from $\mathbf{G}$. Parse $\mathbf{ID}[\mathsf{ID}'] = (i', \mathsf{pk}', \mathsf{ask}')$. Query $\mathbf{Update}(i', \mathsf{Remove})$ and receive apk.
  - For all $\mathsf{ID}'' \in \mathbf{G}$ such that $\mathbf{ID}[\mathsf{ID}'']_3 \neq \bot$:
    - Let $\mathsf{ask} := \mathbf{ID}[\mathsf{ID}'']_3$.
    - $\mathsf{ask} \leftarrow \mathsf{IncDBE.IncSK}(\mathsf{ask}, (i', \mathsf{pk}'), \mathsf{Remove})$.
    - Set $\mathbf{ID}[\mathsf{ID}'']_3 := \mathsf{ask}$.
  - If $q \neq q^*$, sample $K \xleftarrow{R} \{0,1\}^\lambda$ and run $\mathsf{ct} \leftarrow \mathsf{IncDBE.Enc}(\mathsf{apk}, K)$.
  - If $q = q^*$, sample $K, K' \xleftarrow{R} \{0,1\}^\lambda$, query $\mathbf{Challenge}(K, K')$ and receive ct.
  - Set $\mathbf{APK} := \mathsf{apk}, \mathbf{K} := K$, and return $m := (\mathsf{Remove}, i', \mathsf{pk}', \mathsf{apk}, \mathsf{ct})$.
- If $q = \mathbf{PCRefresh}(\mathsf{ID})$:
  - Parse $\mathbf{ID}[\mathsf{ID}] = (i, \mathsf{pk}, \mathsf{ask})$ and $\mathbf{Keys}[j]$ as $(i', \mathsf{pk}', \mathsf{sk}')$.
  - Increment $j \leftarrow j + 1$.
  - Set $\mathbf{ID}[\mathsf{ID}] := (i', \mathsf{pk}', \mathsf{sk}')$.
  - Query $\mathbf{Update}(i, \mathsf{Remove})$ and receive $\mathsf{apk}_1$.
  - For all $\mathsf{ID}'' \in \mathbf{G}$ such that $\mathbf{ID}[\mathsf{ID}'']_3 \neq \bot$:
    - Let $\mathsf{ask} := \mathbf{ID}[\mathsf{ID}'']_3$.
    - $\mathsf{ask} \leftarrow \mathsf{IncDBE.IncSK}(\mathsf{ask}, (i, \mathsf{pk}), \mathsf{Remove})$.
    - Set $\mathbf{ID}[\mathsf{ID}'']_3 := \mathsf{ask}$.
  - Sample $K \xleftarrow{R} \{0,1\}^\lambda$ and run $\mathsf{ct}_1 \leftarrow \mathsf{IncDBE.Enc}(\mathsf{apk}_1, K)$.
  - Query $\mathbf{Update}(i', \mathsf{Add})$ and receive $\mathsf{apk}_2$.
  - For all $\mathsf{ID}'' \in \mathbf{G}$ such that $\mathbf{ID}[\mathsf{ID}'']_3 \neq \bot$:
    - Let $\mathsf{ask} := \mathbf{ID}[\mathsf{ID}'']_3$.
    - $\mathsf{ask} \leftarrow \mathsf{IncDBE.IncSK}(\mathsf{ask}, (i', \mathsf{pk}'), \mathsf{Add})$.
    - Set $\mathbf{ID}[\mathsf{ID}'']_3 := \mathsf{ask}$.
  - If $q \neq q^*$, sample $K \xleftarrow{R} \{0,1\}^\lambda$ and run $\mathsf{ct}_2 \leftarrow \mathsf{IncDBE.Enc}(\mathsf{apk}_2, K)$.
  - If $q = q^*$, sample $K, K' \xleftarrow{R} \{0,1\}^\lambda$, query $\mathbf{Challenge}(K, K')$ and receive $\mathsf{ct}_2$.
  - Set $\mathbf{APK} := \mathsf{apk}_2, \mathbf{K} := K$, and return

$$\mathsf{pp} := (i', \mathsf{pk}'), \quad m := ((\mathsf{Remove}, i, \mathsf{pk}, \mathsf{apk}_1, \mathsf{ct}_1), (\mathsf{Add}, i', \mathsf{pk}', \mathsf{apk}_2, \mathsf{ct}_2)).$$

- If $q = \mathbf{Corrupt}(\mathsf{ID})$:

- Parse $\mathbf{ID}[\mathsf{ID}] = (i, \mathsf{pk}, \mathsf{ask})$.
- Return $\mathsf{st} := (\mathsf{ID}, (i, \mathsf{pk}), \mathbf{APK}, \mathsf{ask}, \mathbf{K})$.[13]

At this point, $\mathcal{B}$ has answered all of $\mathcal{A}$'s queries, and it forwards $\mathcal{A}$'s output $b'$ as its own output. By construction, we have that when $\mathcal{B}$'s challenger has input $b$, the induced view of $\mathcal{A}$ is *identical* to its view in the real CGKA game with bit $b$. Indeed, when $\mathcal{B}$'s challenger has input $b = 0$, the shared CGKA key at epoch $t$ is $K$, and $K$ is the output of **Challenge**. When $b = 1$, the shared CGKA key at epoch $t$ is $K'$, but $K$ (which is independently sampled) is the output of **Challenge**. Thus, it remains to confirm that $\mathcal{B}$ is an admissible adversary according to Definition 6.2, which can be argued as follows.

First, note that by construction, $\mathcal{B}$ makes all **Corrupt** queries to its challenger at the very beginning, i.e., not during the Post-challenge phase. So it remains to check that the set $\{\mathbf{ID}[\mathsf{ID}]_1 : \mathsf{ID} \in \mathbf{G}\}$ for $\mathbf{G}$ at epoch $t$ (corresponding to the set of counters associated with the challenge ciphertext in the IncDBE security game) has no intersection with $T^*$ (corresponding to the set $C$ of corrupted counters in the IncDBE security game). Because $\mathcal{A}$ is admissible, it holds that for all $i \in \{\mathbf{ID}[\mathsf{ID}]_1 : \mathsf{ID} \in \mathbf{G}\}$, $i$ has not been previously corrupted. Indeed, if ID was previously corrupted, there must have been more recently a **PCRefresh**(ID) query that sampled a fresh and uncorrupted $(i, \mathsf{pk})$. Finally, it also holds that $i$ will not be corrupted later, since **Corrupt** queries are not allowed after $t$. □

## 6.3 CGKA with PCS + FS from Updatable DBE

In this section, we show how to use an updatable distributed broadcast encryption scheme (Section 5) to construct a CGKA scheme with post-compromise security and forward security.

**Theorem 6.7.** UpdDBE (Definition 5.1) implies CGKA satisfying correctness (Definition 6.1) and security with respect to the PCS + FS set of admissible queries (Definitions 6.3 and 6.4). Efficiency of all algorithms matches that of the construction from Section 6.2, except that FSRefresh (which was not previously defined) and Process for an FSRefresh query are both linear in the group size.

*Proof.* Note that UpdDBE has the same syntax as IncDBE except for two new algorithms UpdDBE.UpdPK and UpdDBE.UpdSK. The construction is mostly the same as the construction from Section 6.2 (with the UpdDBE algorithms replacing the analogous IncDBE algorithms), except that we add an FSRefresh algorithm along with a branch in Process for processing FSRefresh messages. Other than the new algorithms, the only difference is that st additionally contains an entry for sk (in addition to ID, pp, apk, ask, $K$).

The new algorithms are defined as follows.

- FSRefresh(st, $\mathbf{G}$):
  - Parse $\mathsf{st} = (\mathsf{ID}, \mathsf{pp} = (i, \mathsf{pk}), \mathsf{sk}, \mathsf{apk}, \mathsf{ask}, K)$.
  - Let $S$ be the set of $j$ such that there exists ID such that $(\mathsf{ID}, (j, \mathsf{pk}_j)) \in \mathbf{G}$.
  - Run $(\mathsf{ctref}, \mathsf{apk}, S) \leftarrow \mathsf{UpdDBE.UpdPK}(\mathsf{apk}, S)$ and set $\mathsf{pk} = S[i]$.
  - Run $(\mathsf{ask}, \mathsf{pk}, \mathsf{sk}, S) \leftarrow \mathsf{UpdDBE.UpdSK}(\mathsf{ask}, \mathsf{ctref}, S, \mathsf{pk}, \mathsf{sk})$.
  - Sample $K \xleftarrow{R} \{0,1\}^\lambda$ and run $\mathsf{ct} \leftarrow \mathsf{UpdDBE.Enc}(\mathsf{apk}, K)$.
  - Output

$$\mathsf{st} = (\mathsf{ID}, (i, \mathsf{pk}), \mathsf{sk}, \mathsf{apk}, \mathsf{ask}, K), \quad K, \quad m = (\mathsf{FSRefresh}, i, \mathsf{pk}, \mathsf{apk}, \mathsf{ctref}, \mathsf{ct}), \quad \{\mathsf{ID}, (j, S[j])\}_{\mathsf{ID} \in \mathbf{G}}.$$

- FSRefresh branch of Process(st, $m$, $\mathbf{G}$):
  - Parse $\mathsf{st} = (\mathsf{ID}, \mathsf{pp} = (i, \mathsf{pk}), \mathsf{sk}, \mathsf{apk}, \mathsf{ask}, K)$ and $m = (\mathsf{FSRefresh}, i, \mathsf{pk}, \mathsf{apk}, \mathsf{ctref}, \mathsf{ct})$.
  - Let $S$ be the set of $j$ such that there exists ID such that $(\mathsf{ID}, (j, \mathsf{pk}_j)) \in \mathbf{G}$.

[13] If $\mathbf{D}[\mathsf{ID}] = 1$, $\mathcal{B}$ will actually return a history of states including all of the ask back until when the no-delete flag was first set. This sequence is possible for $\mathcal{B}$ to compute starting from the initial sk associated with ID that it received from its **Corrupt** query.

- Run $(\mathsf{ask}, \mathsf{pk}, \mathsf{sk}, S) \leftarrow \mathsf{UpdDBE.UpdSK}(\mathsf{ask}, \mathsf{ctref}, S, \mathsf{pk}, \mathsf{sk})$.
- Run $K \leftarrow \mathsf{UpdDBE.Dec}(\mathsf{ask}, \mathsf{ct})$.
- Output

$$\mathsf{st} = (\mathsf{ID}, (i, \mathsf{pk}), \mathsf{sk}, \mathsf{apk}, \mathsf{ask}, K), \quad K.$$

Now, we show that given any sequence $(q_1, \ldots, q_\ell)$ of admissible queries according to Definition 6.3 and any PPT adversary $\mathcal{A}$ such that

$$\left| \Pr[\mathcal{A}(\mathsf{crs}) \to b] - \frac{1}{2} \right| = \mathsf{non\text{-}negl}(\lambda),$$

there exists an adversary $\mathcal{B}$ with $\mathsf{non\text{-}negl}(\lambda)$ advantage in the updatable DBE security game (Definition 5.9). By a standard hybrid argument, it suffices to consider sequences $(q_1, \ldots, q_\ell)$ with exactly one **Challenge** query. Assume that there exist $i, j \in [\ell]$ with $q_i = \mathbf{Corrupt}(\mathsf{ID})$, $q_j = \mathbf{Challenge}(t)$ and $\mathsf{q2e}(q_i) > t$, as otherwise security reduces to the PCS-only case analyzed in Section 6.2. Let $t$ be the epoch number that is input to the **Challenge** query, let $q^*_t = \mathsf{e2q}(t)$ be the query to **Add**, **Remove**, **PCRefresh**, or **FSRefresh** that increments $t-1$ to $t$, and let $q^*_{t+1} = \mathsf{e2q}(t+1)$ be the query to **FSRefresh** that increments $t$ to $t+1$ (we know this is a query to **FSRefresh** by admissibility and our assumption).

We define $\mathcal{B}$ as follows. It maintains an (initially empty) map $\mathbf{Keys}\colon j \to (i, \mathsf{pk}, \mathsf{sk})$, an (initially empty) map $\mathbf{ID}\colon \mathsf{ID} \to (i, \mathsf{pk}, \mathsf{sk}, \mathsf{ask})$, an (initially empty) set $\mathbf{G}$ of ID, and variables $\mathbf{APK}$ and $\mathbf{K}$. We will sometimes refer to the three values of $\mathbf{Keys}[j]$ as $\mathbf{Keys}[j]_1, \mathbf{Keys}[j]_2, \mathbf{Keys}[j]_3$, and the four values of $\mathbf{ID}[\mathsf{ID}]$ as $\mathbf{ID}[\mathsf{ID}]_1, \mathbf{ID}[\mathsf{ID}]_2, \mathbf{ID}[\mathsf{ID}]_3, \mathbf{ID}[\mathsf{ID}]_4$. Let $T$ be the total number of **Init** and **PCRefresh** queries among $(q_1, \ldots, q_\ell)$, equivalently the number of times that the Init algorithm is run. Let $T^* \subseteq [T]$ be the subset of $j \in [T]$ such that the $j$'th call to Init results in public parameters pp with the following property: There exists a $\mathbf{Corrupt}(\mathsf{ID})$ query before $q^*_{t+1}$ such that pp is associated with ID at the time of this query. This can be determined by inspecting the sequence $(q_1, \ldots, q_\ell)$. $\mathcal{B}$ begins by doing the following.

For $j \in [T]$:

- Sample $i \xleftarrow{R} [2^\lambda]$, query $\mathbf{KeyGen}(i)$ and receive pk.
- If $j \in T^*$, query $\mathbf{Corrupt}(i)$, receive sk, and add $j \to (i, \mathsf{pk}, \mathsf{sk})$ to **Keys**.
- Otherwise, add $j \to (i, \mathsf{pk}, \bot)$ to **Keys**.

Next, $\mathcal{B}$ will answer $\mathcal{A}$'s queries as follows, setting counter $j = 1$.

For $q \in [q_1, \ldots, q^*_{t+1}]$:

- If $q = \mathbf{Init}(\mathsf{ID})$:
  - Parse $\mathbf{Keys}[j]$ as $(i, \mathsf{pk}, \mathsf{sk})$. Add $\mathsf{ID} \to (i, \mathsf{pk}, \mathsf{sk}, \mathsf{sk})$ to **ID**.
  - Return $(i, \mathsf{pk})$.
  - Increment $j \leftarrow j + 1$.
- If $q = \mathbf{Add}(\mathsf{ID}, \mathsf{ID}')$:
  - Add $\mathsf{ID}'$ to $\mathbf{G}$. Parse $\mathbf{ID}[\mathsf{ID}'] = (i', \mathsf{pk}', \mathsf{sk}', \mathsf{ask}')$.
  - Query $\mathbf{Update}(i', \mathsf{Add})$ and receive apk. If $\mathsf{ask}' \neq \bot$, then **Update** also returns $\mathsf{ask}_{i'}$ and set $\mathbf{ID}[\mathsf{ID}']_4 := \mathsf{ask}_{i'}$.
  - For all $\mathsf{ID}'' \in \mathbf{G} \setminus \{\mathsf{ID}'\}$ such that $\mathbf{ID}[\mathsf{ID}'']_4 \neq \bot$:
    - Let $\mathsf{ask} := \mathbf{ID}[\mathsf{ID}'']_4$.
    - $\mathsf{ask} \leftarrow \mathsf{UpdDBE.IncSK}(\mathsf{ask}, (i', \mathsf{pk}'), \mathsf{Add})$.
    - Set $\mathbf{ID}[\mathsf{ID}'']_4 := \mathsf{ask}$.

- If $q \neq q_t^*$, sample $K \xleftarrow{R} \{0,1\}^\lambda$ and run $\mathsf{ct} \leftarrow \mathsf{UpdDBE.Enc}(\mathsf{apk}, K)$.
- If $q = q_t^*$, sample $K, K' \xleftarrow{R} \{0,1\}^\lambda$, query $\mathbf{Challenge}(K, K')$ and receive ct.
- Set $\mathbf{APK} \coloneqq \mathsf{apk}$, $\mathbf{K} \coloneqq K$, and return $m \coloneqq (\mathsf{Add}, i', \mathsf{pk}', \mathsf{apk}, \mathsf{ct})$.

- If $q = \mathbf{Remove}(\mathsf{ID}, \mathsf{ID}')$:
  - Remove $\mathsf{ID}'$ from $\mathbf{G}$. Parse $\mathbf{ID}[\mathsf{ID}'] = (i', \mathsf{pk}', \mathsf{sk}', \mathsf{ask}')$. Query $\mathbf{Update}(i', \mathsf{Remove})$ and receive apk.
  - For all $\mathsf{ID}'' \in \mathbf{G}$ such that $\mathbf{ID}[\mathsf{ID}'']_4 \neq \bot$:
    - Let $\mathsf{ask} \coloneqq \mathbf{ID}[\mathsf{ID}'']_4$.
    - $\mathsf{ask} \leftarrow \mathsf{UpdDBE.IncSK}(\mathsf{ask}, (i', \mathsf{pk}'), \mathsf{Remove})$.
    - Set $\mathbf{ID}[\mathsf{ID}'']_4 \coloneqq \mathsf{ask}$.
  - If $q \neq q_t^*$, sample $K \xleftarrow{R} \{0,1\}^\lambda$ and run $\mathsf{ct} \leftarrow \mathsf{UpdDBE.Enc}(\mathsf{apk}, K)$.
  - If $q = q_t^*$, sample $K, K' \xleftarrow{R} \{0,1\}^\lambda$, query $\mathbf{Challenge}(K, K')$ and receive ct.
  - Set $\mathbf{APK} \coloneqq \mathsf{apk}$, $\mathbf{K} \coloneqq K$, and return $m \coloneqq (\mathsf{Remove}, i', \mathsf{pk}', \mathsf{apk}, \mathsf{ct})$.

- If $q = \mathbf{PCRefresh}(\mathsf{ID})$:
  - Parse $\mathbf{ID}[\mathsf{ID}] = (i, \mathsf{pk}, \mathsf{sk}, \mathsf{ask})$ and $\mathbf{Keys}[j]$ as $(i', \mathsf{pk}', \mathsf{sk}')$.
  - Increment $j \leftarrow j + 1$.
  - Set $\mathbf{ID}[\mathsf{ID}] \coloneqq (i', \mathsf{pk}', \mathsf{sk}', \mathsf{sk}')$.
  - Query $\mathbf{Update}(i, \mathsf{Remove})$ and receive $\mathsf{apk}_1$.
  - For all $\mathsf{ID}'' \in \mathbf{G}$ such that $\mathbf{ID}[\mathsf{ID}'']_4 \neq \bot$:
    - Let $\mathsf{ask} \coloneqq \mathbf{ID}[\mathsf{ID}'']_4$.
    - $\mathsf{ask} \leftarrow \mathsf{UpdDBE.IncSK}(\mathsf{ask}, (i, \mathsf{pk}), \mathsf{Remove})$.
    - Set $\mathbf{ID}[\mathsf{ID}'']_4 \coloneqq \mathsf{ask}$.
  - Sample $K \xleftarrow{R} \{0,1\}^\lambda$ and run $\mathsf{ct}_1 \leftarrow \mathsf{UpdDBE.Enc}(\mathsf{apk}_1, K)$.
  - Query $\mathbf{Update}(i', \mathsf{Add})$ and receive $\mathsf{apk}_2$.
  - For all $\mathsf{ID}'' \in \mathbf{G}$ such that $\mathbf{ID}[\mathsf{ID}'']_4 \neq \bot$:
    - Let $\mathsf{ask} \coloneqq \mathbf{ID}[\mathsf{ID}'']_4$.
    - $\mathsf{ask} \leftarrow \mathsf{UpdDBE.IncSK}(\mathsf{ask}, (i', \mathsf{pk}'), \mathsf{Add})$.
    - Set $\mathbf{ID}[\mathsf{ID}'']_4 \coloneqq \mathsf{ask}$.
  - If $q \neq q_t^*$, sample $K \xleftarrow{R} \{0,1\}^\lambda$ and run $\mathsf{ct}_2 \leftarrow \mathsf{UpdDBE.Enc}(\mathsf{apk}_2, K)$.
  - If $q = q_t^*$, sample $K, K' \xleftarrow{R} \{0,1\}^\lambda$, query $\mathbf{Challenge}(K, K')$ and receive $\mathsf{ct}_2$.
  - Set $\mathbf{APK} \coloneqq \mathsf{apk}_2$, $\mathbf{K} \coloneqq K$, and return

$$\mathsf{pp} \coloneqq (i', \mathsf{pk}'), \quad m \coloneqq ((\mathsf{Remove}, i, \mathsf{pk}, \mathsf{apk}_1, \mathsf{ct}_1), (\mathsf{Add}, i', \mathsf{pk}', \mathsf{apk}_2, \mathsf{ct}_2)).$$

- If $q = \mathbf{FSRefresh}(\mathsf{ID})$:
  - If $q = q_{t+1}^*$, run the **Refresh** phase to receive $(\mathsf{ctref}, \mathsf{apk})$ and immediately query **Reveal** to receive $\mathsf{apk}, \{\mathsf{sk}_i, \mathsf{ask}_i\}_{i \in \{\mathbf{Keys}[1]_1, \ldots, \mathbf{Keys}[T]_1\}}$. Exit the loop with all information needed to complete the game. Otherwise, continue.
  - Parse $\mathbf{ID}[\mathsf{ID}'] = (i', \mathsf{pk}', \mathsf{sk}', \mathsf{ask}')$. Query $\mathbf{Update}(i', \mathsf{Refresh})$ and receive $(\mathsf{ctref}, \mathsf{apk})$.
  - Let $S$ be the set of $j$ such that there exists ID such that $(\mathsf{ID}, (j, \mathsf{pk}_j)) \in \mathbf{G}$.
  - For all $\mathsf{ID}'' \in \mathbf{G}$ such that $\mathbf{ID}[\mathsf{ID}'']_4 \neq \bot$:
    - Let $(\mathsf{pk}, \mathsf{sk}, \mathsf{ask}) \coloneqq \mathbf{ID}[\mathsf{ID}'']_2, \mathbf{ID}[\mathsf{ID}'']_3, \mathbf{ID}[\mathsf{ID}'']_4$.

* $\mathsf{ask}, \mathsf{pk}, \mathsf{sk}, S \leftarrow \mathsf{UpdDBE.UpdSK}(\mathsf{ask}, \mathsf{ctref}, S, \mathsf{pk}, \mathsf{sk})$.
* Set $\mathbf{ID}[\mathsf{ID}'']_2 := \mathsf{pk}$, $\mathbf{ID}[\mathsf{ID}'']_3 := \mathsf{sk}$, $\mathbf{ID}[\mathsf{ID}'']_4 := \mathsf{ask}$.

– If $q \neq q_t^*$, sample $K \xleftarrow{R} \{0,1\}^\lambda$ and run $\mathsf{ct} \leftarrow \mathsf{UpdDBE.Enc}(\mathsf{apk}, K)$.

– If $q = q_t^*$, sample $K, K' \xleftarrow{R} \{0,1\}^\lambda$, query $\mathbf{Challenge}(K, K')$ and receive ct.

– Set $\mathbf{APK} := \mathsf{apk}$, $\mathbf{K} := K$, and return

$$m := (\mathsf{FSRefresh}, i', \mathsf{pk}', \mathsf{apk}, \mathsf{ctref}, \mathsf{ct}).$$

- If $q = \mathbf{Corrupt}(\mathsf{ID})$:

– Parse $\mathbf{ID}[\mathsf{ID}] = (i, \mathsf{pk}, \mathsf{sk}, \mathsf{ask})$.

– Return $\mathsf{st} := (\mathsf{ID}, (i, \mathsf{pk}), \mathsf{sk}, \mathbf{APK}, \mathsf{ask}, \mathbf{K})$.[14]

At this point, $\mathcal{B}$ has queried **Reveal** and thus knows $\mathsf{ask}_i$ for all $i$ currently in the group. This information allows it to respond to all of $\mathcal{A}$'s future queries in a straightforward manner. Finally, it receives and forwards $\mathcal{A}$'s output $b'$ as its own output.

By construction, we have that when $\mathcal{B}$'s challenger has input $b$, the induced view of $\mathcal{A}$ is *identical* to its view in the real CGKA game with bit $b$. Indeed, when $\mathcal{B}$'s challenger has input $b = 0$, the shared CGKA key at epoch $t$ is $K$, and $K$ is the output of **Challenge**. When $b = 1$, the shared CGKA key at epoch $t$ is $K'$, but $K$ (which is independently sampled) is the output of **Challenge**. Thus, it remains to confirm that $\mathcal{B}$ is an admissible adversary according to Definition 6.3, which can be argued as follows.

Because $\mathcal{A}$ is admissible, the group $\mathbf{G}$ at epoch $t$ is such that for all $i \in \{\mathbf{ID}[\mathsf{ID}]_1 : \mathsf{ID} \in \mathbf{G}\}$ (corresponding to the set of counters associated with the challenge ciphertext in the UpdDBE security game), $i$ has not been previously corrupted. Indeed, if ID was previously corrupted, there must have been more recently a $\mathbf{PCRefresh}(\mathsf{ID})$ query that sampled a fresh and uncorrupted $(i, \mathsf{pk})$. Moreover, again by admissibility, there cannot be any **Corrupt** queries during epoch $t$, meaning there cannot be a **Corrupt** query after $q_t^*$ (when $\mathcal{B}$ issues **Challenge**) through $q_{t+1}^*$ (after which $\mathcal{B}$ issues **Reveal**). Thus, no $\mathsf{ID} \in \mathbf{G}$ can be corrupted *after* epoch $t$. This confirms that the set of counters associated with $\mathcal{B}$'s challenge ciphertext and its set $C$ of corrupted counters in the UpdDBE game do not intersect. □

## Acknowledgments

We acknowledge the use of Claude for copy-editing and for tightening the formal exposition. David J. Wu is supported by NSF CNS-2140975, CNS-2318701, a Sloan Fellowship, a Microsoft Research Faculty Fellowship, a Google Research Scholar Award, an Amazon Research Award, and a gift from the Stellar Development Foundation. This work was done in part while the authors were visiting the Simons Institute for the Theory of Computing, supported in part by a grant from the UC Noyce Initiative. Yevgeniy Dodis was partially supported by a gift from Google.

## References

[AAB+24] Michael Anastos, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Matthew Kwan, Guillermo Pascual-Perez, and Krzysztof Pietrzak. The cost of maintaining keys in dynamic groups with applications to multicast encryption and group messaging. In *TCC*, 2024.

[ABB10] Shweta Agrawal, Dan Boneh, and Xavier Boyen. Efficient lattice (H)IBE in the standard model. In *EUROCRYPT*, 2010.

[ABG+25] Joël Alwen, Chris Brzuska, Jérôme Govinden, Patrick Harasser, and Stefano Tessaro. Succinct PPRFs via memory-tight reductions. In *CRYPTO*, 2025.

[14] If $\mathbf{D}[\mathsf{ID}] = 1$, $\mathcal{B}$ will actually return a history of states including all of the ask back until when the no-delete flag was first set. This sequence is possible for $\mathcal{B}$ to compute starting from the initial sk associated with ID that it received from its **Corrupt** query.

[ACDT20] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In *CRYPTO*, 2020.

[ACDT21] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In *ACM CCS*, 2021.

[ACEP25] Benedikt Auerbach, Miguel Cueto Noval, Boran Erol, and Krzysztof Pietrzak. Continuous group-key agreement: Concurrent updates without pruning. In *CRYPTO*, 2025.

[AFM24] Joël Alwen, Georg Fuchsbauer, and Marta Mularczyk. Updatable public-key encryption, revisited. In *EUROCRYPT*, 2024.

[AFMR25] Joël Alwen, Georg Fuchsbauer, Marta Mularczyk, and Doreen Riepel. Lattice-based updatable public-key encryption for group messaging. *IACR Cryptol. ePrint Arch.*, 2025, 2025.

[AJM22] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. In *CRYPTO*, 2022.

[AMR25] Damiano Abram, Giulio Malavolta, and Lawrence Roy. Key-homomorphic computations for RAM: fully succinct randomised encodings and more. In *CRYPTO*, 2025.

[BBR+23] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The messaging layer security (MLS) protocol. *RFC*, 9420, 2023.

[BDG+22] Alexander Bienstock, Yevgeniy Dodis, Sanjam Garg, Garrison Grogan, Mohammad Hajiabadi, and Paul Rösler. On the worst-case inefficiency of CGKA. In *TCC*, 2022.

[BDR20] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. In *TCC*, 2020.

[BDT22] Alexander Bienstock, Yevgeniy Dodis, and Yi Tang. Multicast key agreement, revisited. In *CT-RSA*, 2022.

[BGI+01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO*, 2001.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS*, 1993.

[BZ14] Dan Boneh and Mark Zhandry. Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation. In *CRYPTO*, 2014.

[CCG+18] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In *ACM CCS*, 2018.

[CGI+99] Ran Canetti, Juan A. Garay, Gene Itkis, Daniele Micciancio, Moni Naor, and Benny Pinkas. Multicast security: A taxonomy and some efficient constructions. In *IEEE INFOCOM*, 1999.

[CHW25] Jeffrey Champion, Yao-Ching Hsieh, and David J. Wu. Registered ABE and adaptively-secure broadcast encryption from succinct LWE. In *CRYPTO*, 2025.

[CW24] Jeffrey Champion and David J. Wu. Distributed broadcast encryption from lattices. In *TCC*, 2024.

[CW26] Jeffrey Champion and David J. Wu. Distributed monotone-policy encryption for DNFs from lattices. In *EUROCRYPT*, 2026.

[DKW21] Yevgeniy Dodis, Harish Karthikeyan, and Daniel Wichs. Updatable public key encryption in the standard model. In *TCC*, 2021.

[DRS04] Yevgeniy Dodis, Leonid Reyzin, and Adam D. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In *EUROCRYPT*, 2004.

[FWW23] Cody Freitag, Brent Waters, and David J. Wu. How to use (plain) witness encryption: Registered ABE, flexible broadcast, and more. In *CRYPTO*, 2023.

[GHMR18] Sanjam Garg, Mohammad Hajiabadi, Mohammad Mahmoody, and Ahmadreza Rahimi. Registration-based encryption: Removing private-key generator from IBE. In *TCC*, 2018.

[GKPW24] Sanjam Garg, Dimitris Kolonelos, Guru-Vamsi Policharla, and Mingyuan Wang. Threshold encryption with silent setup. In *CRYPTO*, 2024.

[GLWW23] Rachit Garg, George Lu, Brent Waters, and David J. Wu. Realizing flexible broadcast encryption: How to broadcast to a public-key directory. In *ACM CCS*, 2023.

[GPV08] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In *STOC*, 2008.

[HLP22] Calvin Abou Haidar, Benoît Libert, and Alain Passelègue. Updatable public key encryption from DCR: efficient constructions with stronger security. In *ACM CCS*, 2022.

[HLWW23] Susan Hohenberger, George Lu, Brent Waters, and David J. Wu. Registered attribute-based encryption. In *EUROCRYPT*, 2023.

[HPS23] Calvin Abou Haidar, Alain Passelègue, and Damien Stehlé. Efficient updatable public-key encryption from lattices. In *ASIACRYPT*, 2023.

[JMM19] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In *EUROCRYPT*, 2019.

[KMW23] Dimitris Kolonelos, Giulio Malavolta, and Hoeteck Wee. Distributed broadcast encryption from bilinear groups. In *ASIACRYPT*, 2023.

[KRS15] Dakshita Khurana, Vanishree Rao, and Amit Sahai. Multi-party key exchange for unbounded parties from indistinguishability obfuscation. In *ASIACRYPT*, 2015.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC*, 2005.

[SDFV+25] David Soler, Carlos Dafonte, Manuel Fernández-Veiga, Ana Fernández Vilas, and Francisco J Nóvoa. Experimental analysis of efficiency of the messaging layer security for multiple delivery services. *arXiv preprint arXiv:2502.18303*, 2025.

[Wee24] Hoeteck Wee. Circuit ABE with $\mathsf{poly}(\mathsf{depth}, \lambda)$-sized ciphertexts and keys from lattices. In *CRYPTO*, 2024.

[Wee25] Hoeteck Wee. Almost optimal KP and CP-ABE for circuits from succinct LWE. In *EUROCRYPT*, 2025.

[WQZD10] Qianhong Wu, Bo Qin, Lei Zhang, and Josep Domingo-Ferrer. Ad hoc broadcast encryption. In *ACM CCS*, 2010.

[WW25] Hoeteck Wee and David J. Wu. Unbounded distributed broadcast encryption and registered ABE from succinct LWE. In *CRYPTO*, 2025.

[WW26] Brent Waters and David J. Wu. Silent threshold cryptography from pairings: Expressive policies in the plain model. In *EUROCRYPT*, 2026.

# A Zeroizing Attack Against a Candidate Updatable DBE Construction

This section demonstrates that a candidate updatable DBE construction that uses a *common refresh shift* across all users admits a *zeroizing attack*. At a high level, the attack exploits the fact that applying the same update vector to every user preserves linear relations between different users' keys and local states. As a result, an adversary can take differences of ciphertext components across two users to eliminate the fresh encryption randomness and expose linear structure in the underlying LWE error.

More concretely, the refresh procedure shifts every public key by the same value $\mathbf{B}\boldsymbol{\Delta}$. While this appears innocuous—and indeed preserves correctness—it has the unintended consequence that pairwise differences of public keys, secret vectors, and opening information remain invariant across refresh. Combined with the linear structure of the encryption algorithm, this invariance allows an adversary to form carefully chosen linear combinations of ciphertext components in which all terms involving the secret $\mathbf{s}$ cancel, leaving only a publicly computable low-norm linear function of the LWE error vector $\mathbf{e}$.

**Construction A.1** (Common-Shift Variant)**.** Let $\lambda$ be a security parameter and $(n, m, q, \widetilde{\sigma})$ be LWE parameters (which are functions of $\lambda$). Let $\sigma$ be a Gaussian width parameter and $\mathfrak{B}$ be a bound parameter. Let $\hat{m} = 4m^5$ and $\xi = \lfloor q/2 \rfloor$. Let $(\mathsf{Setup}_{\mathsf{com}}, \mathsf{Com}, \mathsf{Ver}, \mathsf{Open})$ denote the algorithms of the sparse matrix commitment scheme from Fact 3.5, and let $H\colon \{0,1\}^{\lambda} \to \{0,1\}^{\hat{m}}$ be a hash function that is modeled as a random oracle [BR93]. We construct a candidate updatable distributed broadcast encryption scheme $\Pi = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{IncPK}, \mathsf{IncSK}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{UpdPK}, \mathsf{UpdSK})$ as follows. The construction coincides with Construction 5.10 except that the refresh algorithms evaluate $H$ on the seed $\boldsymbol{\delta}$ *alone* (rather than on the pair $(\boldsymbol{\delta}, i)$), so that all group members are shifted by the same value; we color this change in green.

- $\mathsf{Setup}(1^{\lambda})$: On input the security parameter $\lambda$, the setup algorithm samples the matrix commitment parameters $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B}) \leftarrow \mathsf{Setup}_{\mathsf{com}}(1^{\lambda})$, together with $\mathbf{A} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n \times m}$ and $\mathbf{p} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^{n}$. It outputs $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$.

- $\mathsf{KeyGen}(\mathsf{pp}, i)$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$ where $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$ and an index $i \in [2^{\lambda}]$, the key-generation algorithm samples $\mathbf{r}_i \xleftarrow{\mathrm{R}} \{0, \ldots, \mathfrak{B} - 1\}^{\hat{m}}$ and computes the verification vector $\mathbf{v}_i = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i)$. It then defines the public key $\mathbf{t}_i := \mathbf{B}\mathbf{r}_i + \mathbf{p} - \mathbf{A}\mathbf{v}_i \in \mathbb{Z}_q^n$ and the initial local opening $\widetilde{\mathbf{z}}_i := \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^{\mathsf{T}} \otimes \mathbf{t}_i, i)$, where $\mathbf{u}_i$ is the $i^{\text{th}}$ standard basis vector. It outputs the public key $\mathsf{pk}_i = \mathbf{t}_i$ and the secret key $\mathsf{sk}_i = (i, \mathbf{r}_i, \widetilde{\mathbf{z}}_i)$.

- $\mathsf{IncPK}(\mathsf{pp}, \mathsf{apk}, (i, \mathsf{pk}_i), \mathsf{op})$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, an aggregate public key $\mathsf{apk} = \widetilde{\mathbf{C}} \in \mathbb{Z}_q^{n \times m}$ (where we interpret $\mathsf{apk} = \perp$ to be the all-zeroes matrix), a public key $\mathsf{pk}_i = \mathbf{t}_i$ for user $i$, and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, the public-update algorithm computes the commitment

$$\mathbf{C}_{(i)} = \mathsf{Com}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^{\mathsf{T}} \otimes \mathbf{t}_i).$$

  If $\mathsf{op} = \mathsf{Add}$, it outputs $\mathsf{apk}' = \widetilde{\mathbf{C}} + \mathbf{C}_{(i)}$. If $\mathsf{op} = \mathsf{Remove}$, it outputs $\mathsf{apk}' = \widetilde{\mathbf{C}} - \mathbf{C}_{(i)}$.

- $\mathsf{IncSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, (i, \mathsf{pk}_i), \mathsf{op})$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, an aggregate secret key $\mathsf{ask}_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*})$, a public key $\mathsf{pk}_i = \mathbf{t}_i$ for user $i$, and an operation $\mathsf{op} \in \{\mathsf{Add}, \mathsf{Remove}\}$, the private-update algorithm proceeds as follows:

  - If $i^* = i$, it returns $\mathsf{ask}_{i^*}$ unchanged. Otherwise, it computes the local opening

$$\mathbf{z}_{(i,i^*)} = \mathsf{Open}(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^{\mathsf{T}} \otimes \mathbf{t}_i, i^*).$$

  - If $\mathsf{op} = \mathsf{Add}$, it outputs $\mathsf{ask}'_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*} + \mathbf{z}_{(i,i^*)})$.
  - If $\mathsf{op} = \mathsf{Remove}$, it outputs $\mathsf{ask}'_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*} - \mathbf{z}_{(i,i^*)})$.

- $\mathsf{Enc}(\mathsf{pp}, \mathsf{apk}, \mu)$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$ where $\mathsf{pp}_{\mathsf{com}} = (\mathbf{W}, \mathbf{R}, \mathbf{B})$, an aggregate public key $\mathsf{apk} = \widetilde{\mathbf{C}}$, and a message $\mu \in \{0, 1\}$, the encryption algorithm samples

$$\mathbf{s} \xleftarrow{\mathrm{R}} \mathbb{Z}_q^n, \quad \mathbf{e} \leftarrow \mathcal{D}_{\mathbb{Z}, \widetilde{\sigma}}^{\hat{m}}, \quad \mathbf{D}_1 \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m} \times m}, \quad \mathbf{d}_2 \xleftarrow{\mathrm{R}} \{0,1\}^{\hat{m}}.$$

If $\|\mathbf{e}\| \geq \sqrt{\lambda}\,\widetilde{\sigma}$, it resets $\mathbf{e} = 0^{\hat{m}}$. It then outputs the ciphertext $\mathsf{ct} = (\mathbf{c}_1, \mathbf{c}_2, c_3)$ where

$$\mathbf{c}_1^\intercal = \mathbf{s}^\intercal \mathbf{B} + \mathbf{e}^\intercal, \quad \mathbf{c}_2^\intercal = \mathbf{s}^\intercal(\mathbf{A} + \widetilde{\mathbf{C}}) + \mathbf{e}^\intercal \mathbf{D}_1, \quad c_3 = \mathbf{s}^\intercal \mathbf{p} + \mathbf{e}^\intercal \mathbf{d}_2 + \xi \cdot \mu.$$

- $\mathsf{Dec}(\mathsf{pp}, \mathsf{ask}_{i^*}, \mathsf{ct})$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, an aggregate secret key $\mathsf{ask}_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*})$, and a ciphertext $\mathsf{ct} = (\mathbf{c}_1, \mathbf{c}_2, c_3)$, the decryption algorithm computes the verification vector $\mathbf{v}_{i^*} = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i^*)$ and outputs
$$\mu' = \lfloor c_3 + \mathbf{c}_1^\intercal(\mathbf{r}_{i^*} - \widetilde{\mathbf{z}}_{i^*}) - \mathbf{c}_2^\intercal \mathbf{v}_{i^*} \rceil_2.$$
- $\mathsf{UpdPK}(\mathsf{pp}, \mathsf{apk}, S)$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, an aggregate public key $\mathsf{apk} = \widetilde{\mathbf{C}}$, and a mapping $S$ from the index of each group member to their public key, the algorithm samples $\boldsymbol{\delta} \xleftarrow{\mathrm{R}} \{0,1\}^\lambda$ and computes the *common* shift $\boldsymbol{\Delta} = H(\boldsymbol{\delta})$, which is applied to *every* group member. For all $j \in [\lambda]$, it computes $\mathsf{ctref}_{(j)} \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{apk}, \delta_j)$ where $\delta_j$ is the $j^{\text{th}}$ component of vector $\boldsymbol{\delta}$. Let $\mathsf{ctref} = \big(\mathsf{ctref}_{(1)}, \ldots, \mathsf{ctref}_{(\lambda)}\big)$. Then, for each $i \in S$, let $\mathbf{t}_i = S[i]$ and set $S'[i] = \mathbf{t}_i + \mathbf{B}\boldsymbol{\Delta}$. Compute
$$\mathsf{apk}' = \sum_{i \in S} \mathsf{Com}\left(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes (\mathbf{t}_i + \mathbf{B}\boldsymbol{\Delta})\right)$$
and output $(\mathsf{ctref}, \mathsf{apk}', S')$.
- $\mathsf{UpdSK}(\mathsf{pp}, \mathsf{ask}_{i^*}, \mathsf{ctref}, S, \mathsf{pk}_{i^*}, \mathsf{sk}_{i^*})$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{com}}, \mathbf{A}, \mathbf{p})$, an aggregate secret key $\mathsf{ask}_{i^*} = (i^*, \mathbf{r}_{i^*}, \widetilde{\mathbf{z}}_{i^*})$, a ciphertext $\mathsf{ctref} = \big(\mathsf{ctref}_{(1)}, \ldots, \mathsf{ctref}_{(\lambda)}\big)$ where $\mathsf{ctref}_{(j)} = \big(\mathbf{c}_{1,(j)}, \mathbf{c}_{2,(j)}, c_{3,(j)}\big)$, a mapping $S$ of user indices to public keys, and a public key $\mathsf{pk}_{i^*} = \mathbf{t}_{i^*}$ together with a secret key $\mathsf{sk}_{i^*} = \big(i^*, \mathbf{r}_{i^*}, \mathbf{z}_{(i^*,i^*)}\big)$ for user $i^*$, the private-refresh algorithm proceeds as follows. Compute the verification vector $\mathbf{v}_{i^*} = \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, i^*)$ and, for each $j \in [\lambda]$,
$$\delta_j = \lfloor c_{3,(j)} + \mathbf{c}_{1,(j)}^\intercal(\mathbf{r}_{i^*} - \widetilde{\mathbf{z}}_{i^*}) - \mathbf{c}_{2,(j)}^\intercal \mathbf{v}_{i^*} \rceil_2,$$
and set $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_\lambda) \in \{0,1\}^\lambda$. For each $i \in S$, write $\mathbf{t}_i = S[i]$ (in particular $\mathbf{t}_{i^*} = S[i^*] = \mathsf{pk}_{i^*}$), compute $\boldsymbol{\Delta} = H(\boldsymbol{\delta})$, and set
$$\mathbf{t}_i' \coloneqq \mathbf{t}_i + \mathbf{B}\boldsymbol{\Delta} \quad \text{and} \quad S'[i] \coloneqq \mathbf{t}_i'.$$
Output $(\mathsf{ask}_{i^*}', \mathsf{pk}_{i^*}', \mathsf{sk}_{i^*}', S')$, where
$$\begin{aligned}
\mathsf{pk}_{i^*}' &= \mathbf{t}_{i^*}', \\
\mathsf{ask}_{i^*}' &= \left(i^*,\ \mathbf{r}_{i^*} + \boldsymbol{\Delta},\ \sum_{i \in S} \mathsf{Open}\left(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes \mathbf{t}_i',\ i^*\right)\right), \\
\mathsf{sk}_{i^*}' &= \left(i^*,\ \mathbf{r}_{i^*} + \boldsymbol{\Delta},\ \mathsf{Open}\left(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_{i^*}^\intercal \otimes \mathbf{t}_{i^*}',\ i^*\right)\right).
\end{aligned}$$

As described, the refresh algorithms in Construction A.1 require time that is linear in the size of the group. However, with further optimization, it is possible to compute the aggregated key $\sum_{i \in S} \mathsf{Com}\left(\mathsf{pp}_{\mathsf{com}}, \mathbf{u}_i^\intercal \otimes (\mathbf{t}_i + \mathbf{B}\boldsymbol{\Delta})\right)$ in sublinear time. Nevertheless, we show below an attack strategy which demonstrates that the scheme above is insecure.

**Linear-cancellation attack from common shifts.** We sketch an attack that exploits the fact that the refresh uses a *single* common shift $\boldsymbol{\Delta} = H(\delta)$ so that $\mathsf{pk}_i' - \mathsf{pk}_i = \mathbf{B}\boldsymbol{\Delta}$ is identical for all $i \in S$.

Fix two distinct users $i = 1, 2$ that remain active across a refresh, and let $\mathbf{v}_1 \coloneqq \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, 1)$ and $\mathbf{v}_2 \coloneqq \mathsf{Ver}(\mathsf{pp}_{\mathsf{com}}, 2)$. Let the adversary obtain both secret keys (or equivalently, the corresponding decryption capabilities) for these two users after refresh. From $\mathsf{UpdSK}$ it holds that the *difference* of secret vectors shifts in a known way:

$$\mathbf{r}_1' - \mathbf{r}_2' = (\mathbf{r}_1 + \boldsymbol{\Delta}) - (\mathbf{r}_2 + \boldsymbol{\Delta}) = \mathbf{r}_1 - \mathbf{r}_2,$$

and similarly $\mathsf{pk}_1' - \mathsf{pk}_2' = \mathsf{pk}_1 - \mathsf{pk}_2$ since both add $\mathbf{B}\boldsymbol{\Delta}$.

Consider a challenge encryption $\mathsf{ct} = (\mathbf{c}_1^\intercal, \mathbf{c}_2^\intercal, c_3)$ produced under the aggregated key $\widetilde{\mathbf{C}}$, where (writing $c_1, c_2$ explicitly)

$$\mathbf{c}_1^\intercal = \mathbf{s}^\intercal \mathbf{B} + \mathbf{e}^\intercal, \qquad \mathbf{c}_2^\intercal = \mathbf{s}^\intercal(\mathbf{A} + \widetilde{\mathbf{C}}) + \mathbf{e}^\intercal \mathbf{D}_1.$$

**Forming the cancellation.** Define the following publicly computable linear forms of the challenge ciphertext:

$$L_1 \coloneqq \mathbf{c}_1^{\mathsf{T}}(\mathbf{r}_1 - \mathbf{r}_2) - \mathbf{c}_1^{\mathsf{T}}(\widetilde{\mathbf{z}}_1 - \widetilde{\mathbf{z}}_2), \qquad L_2 \coloneqq \mathbf{c}_2^{\mathsf{T}}(\mathbf{v}_1 - \mathbf{v}_2).$$

Expanding and using Eq. (3.2),

$$\begin{aligned}
L_2 &= \big(\mathbf{s}^{\mathsf{T}}(\mathbf{A} + \widetilde{\mathbf{C}}) + \mathbf{e}^{\mathsf{T}}\mathbf{D}_1\big)(\mathbf{v}_1 - \mathbf{v}_2) \\
&= \mathbf{s}^{\mathsf{T}}\mathbf{A}(\mathbf{v}_1 - \mathbf{v}_2) \;+\; \mathbf{s}^{\mathsf{T}}\widetilde{\mathbf{C}}(\mathbf{v}_1 - \mathbf{v}_2) \;+\; \mathbf{e}^{\mathsf{T}}\mathbf{D}_1(\mathbf{v}_1 - \mathbf{v}_2) \\
&= \mathbf{s}^{\mathsf{T}}\mathbf{A}(\mathbf{v}_1 - \mathbf{v}_2) \;+\; \mathbf{s}^{\mathsf{T}}\left(\mathbf{t}_1 - \mathbf{B}\widetilde{\mathbf{z}}_1 - \mathbf{t}_2 + \mathbf{B}\widetilde{\mathbf{z}}_2\right) \;+\; \mathbf{e}^{\mathsf{T}}\mathbf{D}_1(\mathbf{v}_1 - \mathbf{v}_2) \\
&= \mathbf{s}^{\mathsf{T}}\mathbf{A}(\cancel{\mathbf{y}_1} - \cancel{\mathbf{y}_2}) \;+\; \mathbf{s}^{\mathsf{T}}\left(\mathbf{B}\mathbf{r}_1 - \cancel{\mathbf{A}\mathbf{v}_1} - \mathbf{B}\mathbf{r}_2 + \cancel{\mathbf{A}\mathbf{v}_2}\right) \;+\; \mathbf{s}^{\mathsf{T}}\mathbf{B}(\widetilde{\mathbf{z}}_2 - \widetilde{\mathbf{z}}_1) \;+\; \mathbf{e}^{\mathsf{T}}\mathbf{D}_1(\mathbf{v}_1 - \mathbf{v}_2) \\
&= \mathbf{s}^{\mathsf{T}}\mathbf{B}(\mathbf{r}_1 - \mathbf{r}_2) \;+\; \mathbf{s}^{\mathsf{T}}\mathbf{B}(\widetilde{\mathbf{z}}_2 - \widetilde{\mathbf{z}}_1) \;+\; \mathbf{e}^{\mathsf{T}}\mathbf{D}_1(\mathbf{v}_1 - \mathbf{v}_2).
\end{aligned}$$

On the other hand,

$$L_1 = (\mathbf{s}^{\mathsf{T}}\mathbf{B} + \mathbf{e}^{\mathsf{T}})(\mathbf{r}_1 - \mathbf{r}_2) - (\mathbf{s}^{\mathsf{T}}\mathbf{B} + \mathbf{e}^{\mathsf{T}})(\widetilde{\mathbf{z}}_1 - \widetilde{\mathbf{z}}_2).$$

Subtracting cancels the $\mathbf{s}^{\mathsf{T}}\mathbf{B}$ terms and we are left with

$$L_2 - L_1 = \mathbf{e}^{\mathsf{T}}\mathbf{D}_1(\mathbf{v}_1 - \mathbf{v}_2) \;+\; \mathbf{e}^{\mathsf{T}}(\widetilde{\mathbf{z}}_1 - \widetilde{\mathbf{z}}_2) \;-\; \mathbf{e}^{\mathsf{T}}(\mathbf{r}_1 - \mathbf{r}_2). \tag{A.1}$$

The only remaining unknown contribution in (A.1) is an *explicit linear structure in the LWE error*:

$$\mathbf{e}^{\mathsf{T}}\Big(\mathbf{D}_1(\mathbf{v}_1 - \mathbf{v}_2) + (\widetilde{\mathbf{z}}_1 - \widetilde{\mathbf{z}}_2) - (\mathbf{r}_1 - \mathbf{r}_2)\Big).$$

At a high level, this violates the intended forward-secrecy intuition: the refresh randomness $\mathbf{\Delta}$ does not "decorrelate" users, so taking differences across two users lets an adversary cancel the $\mathbf{s}^{\mathsf{T}}\mathbf{B}(\cdot)$ component and obtain additional linear information about $\mathbf{e}$ that should remain hidden. Once linear information on short vectors is known, the error vector $\mathbf{e}$ may be recoverable and lead to a concrete distinguishing strategy in the forward-secrecy security game.