On the Worst-Case Inefficiency of CGKA

Alexander Bienstock[*] Yevgeniy Dodis[†] Sanjam Garg[‡] Garrison Grogan[§]
Mohammad Hajiabadi[¶] Paul Rösler[‖]

September 18, 2022

Abstract

Continuous Group Key Agreement (CGKA) is the basis of modern Secure Group Messaging (SGM) protocols. At a high level, a CGKA protocol enables a group of users to continuously compute a shared (evolving) secret while members of the group add new members, remove other existing members, and perform state updates. The state updates allow CGKA to offer desirable security features such as forward secrecy and post-compromise security.

CGKA is regarded as a practical primitive in the real-world. Indeed, there is an IETF Messaging Layer Security (MLS) working group devoted to developing a standard for SGM protocols, including the CGKA protocol at their core. Though known CGKA protocols seem to perform relatively well when considering natural sequences of performed group operations, there are no formal guarantees on their efficiency, other than the $O(n)$ bound which can be achieved by trivial protocols, where n is the number of group numbers. In this context, we ask the following questions and provide negative answers.

1. *Can we have CGKA protocols that are efficient in the worst case?* We start by answering this basic question in the negative. First, we show that a natural primitive that we call Compact Key Exchange (CKE) is at the core of CGKA, and thus tightly captures CGKA's worst-case communication cost. Intuitively, CKE requires that: first, n users non-interactively generate key pairs and broadcast their public keys, then, some other *special* user securely communicates to these n users a shared key. Next, we show that CKE with communication cost $o(n)$ by the special user *cannot* be realized in a black-box manner from public-key encryption, thus implying the same for CGKA, where n is the corresponding number of group members.

An extended abstract [10] of this article appears in the proceedings of TCC 2022.

[*] New York University, abienstock@cs.nyu.edu

[†] New York University, dodis@cs.nyu.edu. Partially supported by gifts from VMware Labs and Algorand Foundation, and NSF grants 1815546 and 2055578.

[‡] UC Berkeley and NTT Research, sanjamg@berkeley.edu. This research is supported in part by DARPA under Agreement No. HR00112020026, AFOSR Award FA9550-19-1-0200, NSF CNS Award 1936826, and research grants by the Sloan Foundation, and Visa Inc. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

[§] garrisonwgrogan@gmail.com

[¶] University of Waterloo, mdhajiabadi@uwaterloo.ca. Work supported by an NSERC Discovery Grant RGPIN/03270-2022.

[‖] New York University, paul.roesler@cs.nyu.edu

2. *Can we realize one CGKA protocol that works as well as possible in all cases?* Here again, we present negative evidence showing that no such protocol based on black-box use of public-key encryption exists. Specifically, we show two distributions over sequences of group operations such that no CGKA protocol obtains optimal communication costs on both sequences.

1 Introduction

Secure Group Messaging (SGM) platforms such as Signal Messenger, Facebook Messenger, WhatsApp, etc., are used by billions of people worldwide. SGM has received lots of attention recently, including from the IETF Messaging Layer Security (MLS) working group [8], which is creating an eponymous standard for SGM protocols. While these protocols' security properties are well documented, understanding their *efficiency* properties remains a central research question.

Continuous Group Key Agreement (CGKA) is at the core of SGM protocols. First formalized in [3], CGKA allows a group of users to continuously compute a shared (evolving) symmetric key. This shared group key is re-computed as users asynchronously add (resp. remove) others to (resp. from) the group, as well as execute periodic state refreshes. CGKA provides very robust security guarantees: it not only requires privacy of group keys from non-members, including the facilitating delivery server (which users send CGKA ciphertexts to, in case other group members are offline), but much more. Even in the event of a state compromise in which a user's secret state is leaked to an adversary, group keys should shortly become private again through ordinary protocol state refreshes. Furthermore, in face of such a state compromise, past group keys should remain secure. The former security requirement is referred to as *post-compromise security* (PCS), while the latter is referred to as *forward secrecy* (FS).

Ideally, for use in practice, CGKA protocols should use simple, well-established, and efficient cryptographic primitives and have $O(\log n)$ communication per operation (or at most sub-linear), where n is the number of group members. Indeed, many CGKA protocols in the literature described below claim to have "fair-weather" $O(\log n)$ communication, meaning that when conditions are *good*, communication cost per operation is $O(\log n)$. Such informal claims have pleased practitioners and supported their beliefs that CGKA can be used in the real-world. However, no such formal efficiency guarantees, nor any non-trivial definitions of such *good* conditions have ever been established. Indeed, as elaborated upon below, there are no formal analyses showing that a CGKA protocol can do any better than the trivial $O(n)$ communication cost per operation, on any non-trivial sequence of operations.

CGKA protocols in the literature. Many CGKA protocols have been introduced in the literature to provide the above security properties. The largest portion of these are based on a basic tree structure, as in the Asynchronous Ratchet Tree (ART) protocol [22] and the TreeKEM family of protocols [3, 9, 6, 5, 4, 11, 1], the simplest of which is currently used in MLS [8]. Most of these tree-based protocols are of the same approximate form (although they have slightly different efficiency profiles; see [6] for a comparison based on simulations): each node contains a Public Key Encryption (PKE) key pair, users are assigned to the leaves and only store the secret keys on the path from their leaf to the root, and the root is the group secret. When a user executes an operation, they refresh the secret keys along the path(s) of one (or more) leaves to the root, encrypting these secrets to the siblings along the path(s). Thus, in *very specific* good conditions, communication can easily be seen to be $O(\log n)$. However, due to PCS requirements (elaborated on below), the trees in all of these protocols may periodically *degrade*, resulting in $\Omega(n)$ communication complexity in the worst case, even amortized over many operations.

Instead of using a tree structure, Weidner *et al.* suggest using pairwise channels of the Continuous Key Agreement scheme derived from the famous two-party Signal Secure Messaging protocol [36, 30, 2, 21, 32, 14, 18]. However, this trivial construction of course requires $\Omega(n)$ communication per operation.

In summary, all known CGKA protocols (based on public-key encryption) achieve the same worst-case efficiency as the trivial protocol.

1.1 Our Results

In this paper, we work towards understanding the possible efficiency guarantees that *any* CGKA protocol can achieve in the worst-case, i.e., in cases when the conditions are *not good.* We start by asking the following question:

Can we construct a CGKA protocol that does better than the trivial CGKA protocol in the worst-case?

We provide a negative answer to the above question. In particular, we show that every CGKA (from PKE) has large $\Omega(n)$ worst-case communication cost. Although one can hope that this worst-case will not occur often in practice, until there are better, well-defined assumptions on the structure of operation sequences under which practitioners hope that good efficiency bounds can be proven, there is always a danger of bad efficiency in some cases. As the first step of this lower bound, we show that a natural primitive which we call *Compact Key Exchange* (CKE) is at the core of CGKA, and in fact tightly captures the worst-case communication cost of CGKA. The heart of our negative result is then a black-box separation showing that PKE are insufficient for efficiently realizing CKE. Finally, using the above equivalence, we translate this result into the aforementioned lower bound on CGKA.

Given that no CGKA protocol can be efficient in the worst case, we ask:

Can we realize one CGKA protocol that works as well as possible in all cases?

Here again, we present negative evidence showing that no such protocol based on black-box use of PKE exists. Specifically, we show two distributions over sequences of group operations such that no single CGKA protocol making only black-box use of PKE obtains optimal communication costs on both sequences. That is, any CGKA protocol which acts well on one distribution of operations must have much worse $\Omega(n)$ communication cost on the other distribution; otherwise, it violates our CKE lower bound.

1.2 Compact Key Exchange

To prove our CGKA lower bound, we first isolate and define *Compact Key Exchange* (CKE), a novel primitive that captures one type of scenario that results in large CGKA communication. CKE is related to Multi-Receiver Key Encapsulation Mechanisms [34]. It involves n users who each non-interactively broadcasts a public key, and another special user who sends those n users an encryption of a symmetric key, which only the n users can decrypt. As explained below, we will show that CKE is *equivalent* to CGKA, in terms of worst-case communication complexity.

1.3 Standard Security of Continuous Group Key Agreement

Our CGKA lower bound focuses on the efficiency ramifications of post-compromise security (PCS). The standard form of PCS required for CGKA in the literature [3, 6, 22, 1] is in fact quite strong. Informally, it requires the following two properties:

1. *Double-join prevention.* A malicious user may memorize randomness used in operations they execute. If they are removed from the group at a later time, they must be prevented from using this memorized randomness to *re-join* the group without invitation.

2. *Resilience to randomness leakage.* An honest user's malfunctioning device may continuously leak randomness which the user samples for CGKA operations (e.g., due to implementation flaws or an installed virus). Once the leakage is stopped (due to updating the implementation or removing the virus) and the user performs a state update, the adversary must be prevented from using the previously leaked randomness to obtain future group secrets.

Thus, once a user is removed, all group secrets should be independent of any randomness sampled by them. Similarly, if a user executes a state refresh, all new group secrets should be independent of any randomness *previously* sampled by them.

We emphasize that while the two properties above are rather strong, weakening PCS to exclude them (i.e., where we assume randomness is never leaked and securely deleted after each operation) yields many trivial CGKA protocols (from any PKE) with $O(\log n)$ worst-case communication. For example, one can simply use Tainted TreeKEM (TTKEM) [6] *without taints.*[1] In all these protocols honest parties need to sample secrets for other parties, and are then trusted to delete them once communicated (encrypted) to these other parties. Clearly, most real-world implementations should not be comfortable with this level of trust, and should especially strive for property 1 above instead. Indeed, from very early on in the MLS standardization initiative, requiring property 1 was deemed important[2] and ultimately prioritized over efficiency[3] in the version of TreeKEM used by MLS [8, §13.1]. This protocol, as well as other existing protocols, such as TTKEM, explicitly prevent double-joins (e.g., by sometimes *blanking* or *tainting* nodes that are not on the path to the root from the leaf of a user that is executing an operation) at great efficiency cost; $\Omega(n)$ in the worst-case. Moreover, even though property 2 may seem especially strong, all CGKA definitions in the literature require both properties [3, 6, 22, 1], and our lower bound holds for both (in isolation). Nevertheless, we leave it as an interesting topic for future work to study what kind of efficiency guarantees can be obtained in a more restricted setting, where property 2 is not required.

1.4 Equivalence of CKE and CGKA Worst-Case Communication Complexity

The first step in proving our $\Omega(n)$ CGKA lower bound (from PKE) is showing that CKE and CGKA with the standard PCS notion detailed above are equivalent, both in terms of implication and worst-case communication complexity. It is important to note that in all our definitions of CKE and CGKA, we specify the weakest correctness and security requirements under which our lower

[1] We give a more detailed summary of TTKEM in Appendix F.

[2] First proposal of the TreeKEM design with a discussion about the double-join problem: `https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8/`

[3] Proposal to prevent double-joins in TreeKEM, resulting in linear complexity in the worst-case: `https://mailarchive.ietf.org/arch/msg/mls/Zzw2tqZC1FCbVZA9LKERsMIQXik/`

bounds hold. This only *strengthens* our lower bound. For example, we only consider non-adaptive, passive adversaries.

CKE is at the Core of CGKA. In Section 3, we show that CGKA implies CKE and furthermore that the worst-case communication complexity of CKE from black-box PKE lower bounds that of CGKA from the same primitives. The intuition is as follows. Consider a CGKA group with n members at a certain time during its lifetime. To ensure that our lower bound is meaningful, we allow for any sequence of operations to be executed up until this point. Now, consider the situation in which user A adds k new users. If the CGKA protocol only uses PKE, then each added user only stores secrets (besides their own) that were generated by user A.[4] If user B removes user A as the next operation, then due to PCS, every secret which the k added users shared with any of the current group members cannot be re-used; user A must have generated all of them and thus could potentially (maliciously) re-join the group without being added if one of the secrets is reused. Thus, as part of the remove operation, user B must communicate the new group key to each of the other k added users, with only the knowledge of their (independent) public keys. This is exactly the setting of CKE. Indeed if $k = \Omega(n)$, and additionally we can show that the ciphertext size for CKE must be $\Omega(n)$, then we can show the same for when user B removes user A in CGKA above. Furthermore, if user C then removes user B, we are in the same situation again, and thus this ciphertext must also be $\Omega(n)$. We can repeat this scenario *ad infinitum*, where after a user executes a remove in the sequence, they add a new user, such that even amortized over a long sequence of operations, the communication cost is $\Omega(n)$. We in fact further generalize this result in Section 3 to intuitively show that if α users add the k new users then execute ℓ rounds of sequential state refreshes, the combined communication cost of each round is $\Omega(k)$.

A bit more formally, we show how to construct CKE for k users from CGKA in a manner such that if the CGKA ciphertext is small for the above operation and the CGKA protocol only uses PKE in a black-box manner, then the corresponding CKE ciphertext is small, contradicting our lower bound for CKE, discussed below.

Difference from lower bound of [11]. It is important to mention that our CGKA communication complexity lower bound already holds for fully synchronous, non-concurrent CGKA executions. Hence, the lower bound by Bienstock *et al.* [11] that uses symbolic proof techniques to show a communication lower bound for concurrently initiated operations in CGKA executions (with required fast PCS recovery[5]) is entirely independent with respect to our employed methods and resulting statement.

CKE tightly implies CGKA. For completeness, in Appendix E we also show that one can use CKE to construct a CGKA protocol where the worst-case communication complexity of the CGKA protocol is proportional to that of the used CKE protocol. The CGKA protocol simply lets the user, executing a given CGKA operation, run the CKE algorithm of the special CKE user to communicate a fresh group key to the public keys of all current CGKA group members. Therefore,

[4] Note: for any CGKA protocol, it could be that each of the added k users may share secrets with all of the current group members, derived from non-interactive key exchange using key-bundles stored on a server. However, these shared secrets are only between pairs of users, and thus do not seem useful for establishing the group secret (since secure communication between pairs of users can already be achieved via PKE).

[5] Unlike in [1] who circumvent the [11] lower bound by allowing for slower PCS recovery.

CGKA and CKE are surprisingly equivalent in terms of both cryptographic strength *and* worst-case (communication) complexity; f one could construct CKE efficiently, they could also construct CGKA efficiently, and vice versa.

1.5 Black-Box Compact Key Exchange Lower Bound

In order to prove the CGKA lower bounds discussed above, we need a lower bound on the underlying CKE primitive. Therefore, in Section 4, we prove a black-box separation showing that all CKE protocols that make black-box use of public-key encryption (PKE) require the ciphertext sent from the special user to the n users to have size $\Omega(n)$, *irrespective of the sizes of the public keys* that the n users have sent to the special user. Our impossibility holds even if the scheme comes with a CRS, of arbitrary size. Ruling out schemes that allow for a CRS will help us with our CGKA lower bounds.

Intuitively, since the n public keys are generated *independently* from each other, our result implies that there is no non-trivial "compression" operation that the special user can do to save over the trivial protocol: choosing a key and separately encrypting the key to each user independently.

Relations to broadcast encryption. We note that the notion of CKE is incomparable to that of broadcast encryption, at least in an ostensible sense. Recall that a broadcast encryption scheme is a type of attribute-based encryption that allows for broadcasting a message to a subset of users, in a way that the resulting ciphertext is compact. One crucial difference between broadcast encryption and CKE is that under CKE, users have independent secret keys, while under broadcast encryption, user secret keys are correlated, all obtained via a master secret key.

Relations to other black-box impossibility results. The work of Boneh et al. [15] shows that identity-based encryption (IBE) is black-box impossible from trapdoor permutations (TDPs). A striking similarity between IBE and CKE is that both deal with some form of compactness: that of public parameters (PP) for IBE and of ciphertexts in CKE. The techniques of [15] crucially rely on the number of identities being much larger than the number of queries required to generate a public parameter. In our setting, this is no longer the case: the number of queries made by the encryption algorithm to generate a compact ciphertext may be much larger than n, and hence the techniques of [15] do not work in our setting. In addition, we allow the CRS to grow with the number of identities.

Extensions and limitations of our impossibility results. We believe that out impossibility should extend quite naturally to separate CKE from trapdoor permutations (TDPs), though we have not worked out the details. Our impossibility results have no bearing on the base primitive being used in a non-black-box way, and indeed by using strong tools such as indistinguishability obfuscation (which inherently results in non-black-box constructions), one might be able to build compact CKE.

Overview. Our impossibility result is proved relative to a random PKE oracle $\mathbf{O} := (\mathbf{g}, \mathbf{e}, \mathbf{d})$. We give an attack against any CKE protocol (CRSGen, Init, Comm, Derive) (Definition 4.2) instantiated with $\mathbf{O}$. To give some intuition about the attack, suppose $\mathbf{e}$ is an encryption oracle, whose output length (i.e., the ciphertext length) is sufficiently larger than its input length (i.e., the length of (pk, m, r)). This in particular implies that in order to get a valid (pk, c)—one under which there exists some m and r such that $\mathbf{e}(\mathsf{pk}, m, r) = c$—one has to call the $\mathbf{e}$ oracle first. Now if a CKE ciphertext for n users has length $o(n)$, this means that one can "embed" at most $o(n)$ valid $\mathbf{e}$-ciphertexts into C. Say the ciphertexts are $c_1, \ldots, c_t$ with corresponding public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_t$, where $t \in o(n)$. This means that we need at most t effective trapdoors (with respect to $\mathbf{O}$) to decrypt C, namely the trapdoors that correspond to $(\mathsf{pk}_1, \ldots, \mathsf{pk}_t)$. Also, since C should be decryptable by each user, the set of "effective" trapdoors for each user (those required to decrypt C) should be a subset of all these t trapdoors. Now since $t = o(n)$, there exists a user whose effective trapdoors are a subset of all other users. But since the CKE secret keys for all users are generated independently and with no correlations, if we run the CKE key generation algorithm many times, we should be able to recover all the required trapdoors, for at least one user. This is the main idea of the proof.

The above overview is overly simplistic, omitting many subtleties. For example, an $\mathbf{e}$-ciphertext that is decrypted may come from one of the public keys $\mathsf{PK}_1, \ldots, \mathsf{PK}_n$ (which can be arbitrarily large), and not from C itself. Second, the notion of "embedded ciphertexts" in C is not clear. We will formalize all these subtleties in Section 4 and will give a more detailed overview there, after establishing some notation.

New techniques. Our proofs introduce some techniques that may be of independent interest. Firstly, our proofs involve oracle sampling steps (a technique also used in many other papers), but one novel thing in our proofs is that we need to make sure that the sampled oracles do not contain a certain set of query/response pairs. In comparison, prior oracle sampling techniques involve choosing oracles that agree with a set of query/answer pairs. This technique of making certain query/response pairs off-limits, and the implications proved, might find applications in proving other impossibility results. Moreover, our proofs use theorems about non-uniform attacks against random oracles [23, 20] to argue that an $o(n)$ CKE ciphertext cannot embed n ciphertexts; we find this connection novel.

In Section 4, we will give an overview (and the proof) for the restricted construction setting in which oracle access is of the form $(\mathrm{CRSGen}^{\mathbf{g}}, \mathrm{Init}^{\mathbf{g}}, \mathrm{Comm}^{\mathbf{e}}, \mathrm{Derive}^{\mathbf{d}})$. This will capture most of the ideas that go into the full proof. We will then give a proof for the general construction case in Section C.

1.6 No *Single* Optimal CGKA Protocol Exists

In Section 5, we present another negative result for CGKA protocols that make black-box use of PKE. Naturally, CGKA protocols proceed in an online manner such that users do not know which operations will be executed next. Therefore, users have to make choices when executing operations that may result in unnecessary communication. We leverage this situation to show that there does not exist any *single* CGKA protocol that makes black-box use of PKE and that has optimal communication costs for every sequence that may be executed. More specifically, for every CGKA protocol Π, there exists some distribution of CGKA operations Seq and some other CGKA protocol Π' such that Π has much higher communication costs than Π' when executing Seq.

Our driving example is as follows: suppose again that starting with a CGKA group in arbitrary state, k users are added by user A and remain offline. Next, α users (including user A) execute state refreshes. In this case, some protocols might use a strategy which, through these state refreshes, create and communicate extra redundant secrets for the k added users, while others may use a strategy which simply relies on those secrets communicated by user A. For the former strategy, if the k added users afterwards come online and execute their own state refreshes, then the communication of these extra secrets will have been unnecessary, and a protocol which follows the latter strategy will have much lower communication cost. However, for the latter strategy, if one of the $\alpha - 1$ users, user j, who only communicated a small amount ($o(k)$) in their state refresh thereafter remains offline while the other $\alpha - 1$ users execute rounds of sequential state refreshes, then we know from what we prove in Section 3 that each of the rounds will have $\Omega(k)$ communication cost. This is intuitively because the k added users mostly share secrets with the $\alpha - 1$ users excluding user j, and thus when these $\alpha - 1$ users perform state refreshes, they must re-communicate secrets to the k added users. On the other hand, a protocol that follows the former strategy can have much lower communication cost if the state refresh ciphertext of user j *alone* was large ($\Omega(k)$). This is intuitively because the k added users still share enough secrets with user j, so that when the other $\alpha - 1$ users execute their state refreshes, they do not need to communicate much new to the added users.

1.7 Lessons Learned for Practice

Our results show that the execution of a CGKA protocol causes impractical communication overhead amongst the group members if (1) the CGKA protocol is built from PKE only, (2) the CGKA protocol achieves the weakest accepted notion of security, and (3) group members of the protocol execution initiate certain non-trivial operation sequences. We note that, on an intuitive level, PKE are essentially the only building blocks of all practical CGKA constructions. Furthermore, all of the non-trivial operation sequences employed for our lower bounds are legitimate, and *could* happen in practice. Consequently, impractical worst-case communication overheads seem to be inevitable. However, in order to avoid such impractical communication overheads, one could (a) try to find suitable practical building blocks *other* than PKE to circumvent the lower bound, (b) lower the security requirements for CGKA (which we strongly advise against!), or (c) identify *all problematic* operation sequences and then forbid their execution. We believe that (a) finding better constructions and (c) identifying all such problematic operation sequences are interesting questions that we leave open for future work. However, for (a), we emphasize that one would ultimately need to circumvent our CKE lower bound. Although one may be able to do so using strong primitives such as indistinguishability obfuscation (as in the multi-party non-interactive key exchange of [16]), we view it as a challenging problem to do so from *practical* tools other than PKE.

We provide some further consequences of our lower bound in practice below.

CGKA with two administrators. Many real-world SGM systems in production may impose membership policies on users. That is, it could be that there are only a few “administrators” that are allowed to add and remove others from the group, while everyone else can only update their state and send messages. As shown by [12], for the setting in which there is only ever *one* administrator, CGKA boils down to the classical setting of Multicast Encryption [35, 37, 28, 17, 29, 33, 13]. Since there is only one administrator in Multicast, $O(\log n)$ communication complexity is easily achieved even with security property 1 above [12] (however, security property 2 already results in $\Omega(n)$ complexity for the administrator in Multicast). This is due to the fact that the sole administrator is never removed and executes all operations; thus she can use a tree as in some of the aforementioned CGKA protocols, and never allow it to degrade.

Therefore, a natural question is: In the setting of two administrators that can replace one another with new administrators, and where only property 1, but not property 2, is required for the administrators; can we retain $O(\log n)$ communication?[6] One can observe that our above lower bound answers this question in the negative. Indeed, there only ever need to be two administrators in the group. If so, then as above, one administrator can add k users, then the second administrator can replace the first with a new third administrator, then the third administrator can replace the second administrator, and so on. Thus, the jump from one to two administrators in the worst case requires communication to increase from $O(\log n)$ to $\Omega(n)$ per operation, if security property 1 (and not 2) is required.

MLS propose-and-commit framework. The latest MLS protocol draft (version 14) [8], uses the "propose-and-commit" framework for CGKA. In this framework, users can publish many messages that *propose* different group operations (adding/removing others or updating their state), and a new group key is not established until some user subsequently publishes a *commit* message. The motivation behind this design is to allow for greater concurrency of CGKA operations: In prior drafts of MLS, users would attempt to establish a new group key with each operation. If many users desired to execute an operation at the same time and published corresponding CGKA ciphertexts, the delivery server would have to choose one such ciphertext to deliver to all group members (and thus only one of the group operations would be executed). With propose-and-commit, the delivery server still has to choose between commit messages, but many proposed group operations can be combined inside a single commit.

We however observe that we can still apply our above CGKA lower bound to this framework. Indeed, consider the scenario wherein one user (resp. administrator) A proposes to add k users, then publishes a commit for these additions. Thereafter, some other user (resp. administrator) B can replace A in a new proposal, then publish a commit for this replacement. Again, replacements can be repeated *ad infinitum*, and it can easily be seen that each such commit will still cost $\Omega(n)$ communication. Hence, our result of Section 3 naturally holds in the propose-and-commit framework.

2 Definitions

In this section, we define syntax and non-adaptive, one-way notions of security for Continuous Group Key Agreement and Compact Key Exchange. First, we introduce some notation.

Notation. For algorithm A, $y \leftarrow A(x; r)$ means that A on input x with randomness r outputs y. If r is not made explicit, it is assumed to be sampled uniformly at random, and we use notation $y \leftarrow_{\$} A(x)$. We will also use the notation $x \leftarrow_{\$} \mathsf{X}$ to denote uniformly random sampling from set X. We will use dictionaries for our CGKA security game. The value stored with key x in dictionary D is denoted by $D[x]$. The statement $D[*] \leftarrow v$ initializes a dictionary D in which the default value for each key is v.

[6] If neither administrator is removed, of course $O(\log n)$ communication can be retained if they share a multicast tree.

2.1 Continuous Group Key Agreement

In the simple, restricted form that we consider here, *Continuous Group Key Agreement* (CGKA) allows a dynamic set of users to continuously establish symmetric group keys. For participating in a group, a user first generates a public key and a secret state via algorithm Gen. With the secret state, a user can add or remove users to or from a group via algorithms Add and Rem. Furthermore, each user can update the secrets in their state from time to time to recover from adversarial state corruptions via algorithm Up. We call the latter three actions *group operations*. After all users process a group operation via algorithm Proc, they share the same group key. In order to analyze the *most efficient* form of CGKA, we assume a central bulletin board $\mathbf{B}$ to which public information on the current group structure is posted (initially empty). Thus, newly added users can obtain the relevant information about the group (which intuitively may be of size $\Omega(n)$ anyway, where n is the current number of group members) from $\mathbf{B}$, instead of receiving it explicitly from the adding user. Note: the MLS protocol specification indeed suggests the added user can obtain the group tree of the protocol (size $\Omega(n)$) from a bulletin board (the delivery server) in this manner [8].

In the following, the added user simply downloads the *entire* board. Of course, in practice, this would be very inefficient, but this only strengthens our lower bound on the amount of communication sent between *current* group members (as opposed to the amount of information retrieved from the bulletin board by added users).

Definition 2.1. *A* Continuous Group Key Agreement *scheme* CGKA = (Gen*,* Add*,* Rem*,* Up*,* Proc) *consists of the following algorithms:*[7]

- Gen *is a PPT algorithm that outputs* $(\mathsf{ST}, \mathsf{PK})$*.*

- Add *is a PPT algorithm that takes in* $(\mathsf{ST}, \mathsf{PK})$*, where* ST *is the secret state of the user invoking the algorithm and* PK *is the public key of the added user, and outputs* (ST', K, C)*, where* ST' *is the updated secret state of the invoking user,* K *is the new shared group key, and* C *is the ciphertext that is sent to (and then processed by) the group members. For efficiency purposes,* $C = (C_G, C_B)$ *consists of a share* C_G *that is sent to all group members* directly *and a share* C_B *that is posted to the central bulletin board* $\mathbf{B}$*.*

- Rem *is a PPT algorithm that takes in* $(\mathsf{ST}, \mathsf{PK})$*, where* ST *is the secret state of the user invoking the algorithm and* PK *is the public key of the removed user, and outputs* (ST', K, C) *as above.*

- Up *is a PPT algorithm that takes in secret state* ST *of the user invoking the algorithm and outputs* (ST', K, C) *as above.*

- Proc *is a deterministic, polynomial time algorithm that takes in* (ST, C_G)*, where* ST *is the secret state of the user invoking the algorithm and* C_G *is the ciphertext* directly *received for an operation, and outputs updated state and group key* (ST', K)*. For users that were just added to the group,* Proc *additionally takes in bulletin board* $\mathbf{B}$*. If the operation communicated via* C *removes the processing user from the group,* K *is set to a special symbol* $\perp$*.*

[7] For the sake of comprehensible communication analysis, we do not provide an explicit Create$(\mathsf{ST}, \mathsf{PK}_1, \ldots, \mathsf{PK}_n)$ algorithm (for which in practice, $\Omega(n)$ ciphertext size could be tolerated). Instead, we require the group creator to one-by-one add $\mathsf{PK}_1, \ldots \mathsf{PK}_n$, which allows us to prove a more meaningful lower bound on just Add, Rem, and Up operations.

Initialization: Set (i) $t = 0$; (ii) $\mathbf{WeakEpochs}, \mathbf{WeakUsers} = \emptyset$; and (iii) $\mathbf{G}[*], \mathbf{Rand}[*], \mathbf{ST}[*], \mathbf{K}[*] \leftarrow \perp$.

- $\mathbf{Gen}()$ executes $(\mathsf{ST}, \mathsf{PK}) \leftarrow_{\$} \mathrm{Gen}()$, sets $\mathbf{ST}[\mathsf{PK}] \leftarrow \mathsf{ST}$, and returns PK.
- $\mathbf{Add}(\mathsf{PK}, \mathsf{PK}^*)$ first aborts if (i) $\mathsf{PK} = \mathsf{PK}^*$; (ii) $t \neq 0$ and $\mathsf{PK} \notin \mathbf{G}[t]$; or (iii) $\mathsf{PK}^* \in \mathbf{G}[t]$. Otherwise it:
    1. For randomly sampled r, sets $\mathbf{Rand}[\mathsf{PK}, t+1] \leftarrow r$ and executes $(\mathbf{ST}[\mathsf{PK}], \mathbf{K}[t+1, \mathsf{PK}], (C_G, C_B)) \leftarrow \mathrm{Add}(\mathbf{ST}[\mathsf{PK}], \mathsf{PK}^*; r)$.
    2. Sets $\mathbf{G}[t+1] \leftarrow \mathbf{G}[t] \cup \{\mathsf{PK}, \mathsf{PK}^*\}$.
    3. For every $\mathsf{PK}' \in \mathbf{G}[t] \setminus \{\mathsf{PK}\}$, executes $(\mathbf{ST}[\mathsf{PK}'], \mathbf{K}[t+1, \mathsf{PK}']) \leftarrow \mathrm{Proc}(\mathbf{ST}[\mathsf{PK}'], C_G)$. Also executes $(\mathbf{ST}[\mathsf{PK}^*], \mathbf{K}[t+1, \mathsf{PK}^*]) \leftarrow \mathrm{Proc}(\mathbf{ST}[\mathsf{PK}^*], C_G, \mathbf{B})$.
    4. If $(\mathbf{WeakUsers} \cap \mathbf{G}[t+1]) \neq \emptyset$, sets $\mathbf{WeakEpochs} \leftarrow \mathbf{WeakEpochs} \cup \{t+1\}$.
    5. Increments $t \leftarrow t+1$ and returns (C_G, C_B).
- $\mathbf{Rem}(\mathsf{PK}, \mathsf{PK}^*)$ first aborts if (i) $t = 0$; (ii) $\mathsf{PK} = \mathsf{PK}^*$; (iii) $\mathsf{PK} \notin \mathbf{G}[t]$; or (iv) $\mathsf{PK}^* \notin \mathbf{G}[t]$. Otherwise, it:
    1. For randomly sampled r, sets $\mathbf{Rand}[\mathsf{PK}, t+1] \leftarrow r$ and executes $(\mathbf{ST}[\mathsf{PK}], \mathbf{K}[t+1, \mathsf{PK}], (C_G, C_B)) \leftarrow \mathrm{Rem}(\mathbf{ST}[\mathsf{PK}], \mathsf{PK}^*; r)$.
    2. Sets $\mathbf{G}[t+1] \leftarrow \mathbf{G}[t] \setminus \{\mathsf{PK}^*\}$.
    3. For every $\mathsf{PK}' \in \mathbf{G}[t] \setminus \{\mathsf{PK}\}$, executes $(\mathbf{ST}[\mathsf{PK}'], \mathbf{K}[t+1, \mathsf{PK}']) \leftarrow \mathrm{Proc}(\mathbf{ST}[\mathsf{PK}'], C_G)$.
    4. If $(\mathbf{WeakUsers} \cap \mathbf{G}[t+1]) \neq \emptyset$, sets $\mathbf{WeakEpochs} \leftarrow \mathbf{WeakEpochs} \cup \{t+1\}$.
    5. Increments $t \leftarrow t+1$ and returns (C_G, C_B).
- $\mathbf{Up}(\mathsf{PK})$ first aborts if (i) $t = 0$; or (ii) $\mathsf{PK} \notin \mathbf{G}[t]$. Otherwise, it:
    1. For randomly sampled r, sets $\mathbf{Rand}[\mathsf{PK}, t+1] \leftarrow r$ and executes $(\mathbf{ST}[\mathsf{PK}], \mathbf{K}[t+1, \mathsf{PK}], (C_G, C_B)) \leftarrow \mathrm{Up}(\mathbf{ST}[\mathsf{PK}]; r)$.
    2. Sets $\mathbf{G}[t+1] \leftarrow \mathbf{G}[t]$ and $\mathbf{WeakUsers} \leftarrow \mathbf{WeakUsers} \setminus \{\mathsf{PK}\}$.
    3. For every $\mathsf{PK}' \in \mathbf{G}[t+1] \setminus \{\mathsf{PK}\}$, executes $(\mathbf{ST}[\mathsf{PK}'], \mathbf{K}[t+1, \mathsf{PK}']) \leftarrow \mathrm{Proc}(\mathbf{ST}[\mathsf{PK}'], C_G)$.
    4. If $(\mathbf{WeakUsers} \cap \mathbf{G}[t+1]) \neq \emptyset$, sets $\mathbf{WeakEpochs} \leftarrow \mathbf{WeakEpochs} \cup \{t+1\}$.
    5. Increments $t \leftarrow t+1$ and returns (C_G, C_B).

- $\mathbf{Corr}(\mathsf{PK})$ first sets $\mathbf{WeakUsers} \leftarrow \mathbf{WeakUsers} \cup \{\mathsf{PK}\}$ and $\mathbf{WeakEpochs} \leftarrow \mathbf{WeakEpochs} \cup \{t' \leq t : \mathsf{PK} \in \mathbf{G}[t']\}$. Then it returns $\mathbf{ST}[\mathsf{PK}]$ and $\mathbf{Rand}[\mathsf{PK}, t']$, for every $t' \leq t$.

Figure 1: The CGKA correctness and security games.

Correctness and Security. We define correctness and security of CGKA via games that are played by an adversary $\mathcal{A}$, in which $\mathcal{A}$ controls an execution of the CGKA protocol. For simplicity and clarity, we only consider a *non-adaptive* protocol execution in a *single group*. The games are specified in Figure 1.

Before either game starts, the adversary specifies the sequence of queries to the oracles $\mathbf{Gen}()$, $\mathbf{Add}()$, $\mathbf{Rem}()$, $\mathbf{Up}()$, and $\mathbf{Corr}()$ that will be executed. $\mathbf{Gen}()$ allows the adversary to initialize a new user, from which it receives the corresponding public key PK. The other oracles allow the adversary to execute group operations, i.e., to add, remove, and update users, respectively. Additionally, for the security game, the adversary beforehand specifies the *epoch* t which it will attack, i.e., for which it will guess the group key. The game starts in epoch $t = 0$, then increments t each time a group operation oracle is queried. The game forces the adversary to first query $\mathbf{Add}()$ to initialize the group. It keeps track of group members for each epoch using dictionary G. For simplicity, in each group operation query, the game immediately uses each current group member's state to process the resulting ciphertext directly sent to them, C_G, along with the current bulletin board $\mathbf{B}$, in the case of an added user. Dictionary $\mathbf{K}$ keeps track of the group key that each user computes for each epoch. Each group operation oracle returns $C = (C_G, C_B)$ to the adversary.

Definition 2.2. *A CGKA scheme* CGKA *is* correct *if for every adversary* $\mathcal{A}$ *against the correctness game defined by Figure 1, and for all* t *and* $\mathsf{PK}, \mathsf{PK}' \in \mathbf{G}[t]$*:* $\Pr[\mathbf{K}[t, \mathsf{PK}] = \mathbf{K}[t, \mathsf{PK}']] = 1$*.*

Our notion of security is slightly weakened compared to the standard definition in the CGKA literature, which only strengthens our lower bound. That is, the corruption of a user may affect the security of those keys that were established in the past while this user was a group member. Thus, forward secrecy is not captured. Also, we do not consider authenticity.[8] However, our notion still captures basic security requirements plus standard PCS requirement (mentioned in the introduction), as explained below.

We first explain the importance of dictionary **Rand**, in addition to sets **WeakEpochs** and **WeakUsers**, which allow the game to capture this security. **Rand** keeps track of the randomness the users sample to execute the operations of each epoch. Intuitively, **WeakEpochs** and **WeakUsers** keep track of those epochs and users that are insecure, respectively. When the adversary queries oracle **Corr**(PK), the game returns the corresponding user's secret state, as well as the randomness which she used to execute *all* of her past group operations. Thus, the game adds PK to **WeakUsers** and since we do not require forward secrecy, it also adds to **WeakEpochs** every past epoch in which the corresponding user was in the group. Now, for every **Up**(PK) query, the game removes PK from **WeakUsers**. This in part captures PCS: in every group operation query, if there are still weak users in the group (i.e., $(\mathbf{WeakUsers} \cap \mathsf{G}[t+1]) \neq \emptyset$), then the game adds the new epoch $t+1$ to **WeakEpochs**. So, if there is a member of the group that was corrupted and did not since update their state, the epoch is deemed weak. Conversely, as soon as every group member updates their state or is removed after a corruption, epochs are no longer deemed weak.

After receiving all return values of the pre-specified sequence's queries to these oracles, the adversary outputs a key K. This key K is a guess for the actual group key established in epoch t, where t is the pre-specified attack epoch. Note that this recoverability definition is weaker than standard indistinguishability definitions, which strengthens our lower bound.

Definition 2.3. *A CGKA scheme* CGKA *is* secure *if for every PPT adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *against the security game defined by Figure 1:*

$$\Pr\left[K \leftarrow_{\$} \mathcal{A}_2(\omega, \mathsf{Trans}) : K = \mathbf{K}[t, \mathsf{PK}^*]; t \notin \mathbf{WeakEpochs}; \mathsf{PK}^* \in \mathsf{G}[t]; (\omega, \mathsf{Seq}, t) \leftarrow_{\$} \mathcal{A}_1(1^\lambda)\ \right] \leq \mathsf{negl}(\lambda),$$

where $\mathcal{A}_1$ *non-adaptively specifies the sequence of oracle queries* Seq *and the attacked epoch* t*, and* $\mathcal{A}_2$ *guesses the attacked key when obtaining the transcript of oracle return values* Trans*.*

2.2 Compact Key Exchange

We can now define Compact Key Exchange with access to a common reference string (CRS). Such protocols allow some users $1, \ldots, n$ to sample independent (across users) key pairs $(\mathsf{SK}_1, \mathsf{PK}_1), \ldots,$

[8] Analyzing the effect of required authenticity under weak randomness [7] on (communication) complexity in the group setting [31], as well as of extended security goals such as anonymity [25] remains an interesting open question.

$(\mathsf{SK}_n, \mathsf{PK}_n)$, then publicly broadcast $\mathsf{PK}_1, \ldots, \mathsf{PK}_n$. Upon reception of these public keys, special user 0 generates a key K and message C, and broadcasts C. Finally, upon reception of C, every user $i \in [n]$ uses SK_i, the set of public keys $\{\mathsf{PK}_j\}_{j\in[n]}$, and C to derive K.

Definition 2.4. *A* Compact Key Exchange *scheme* $\mathsf{CKE} = (\mathrm{CRSGen}, \mathrm{Init}, \mathrm{Comm}, \mathrm{Derive})$ *in the standard model with common reference string* $\mathsf{CRS} \in \mathcal{CRS}$ *consists of the following algorithms:*

- Init *is a PPT algorithm that takes in* $\mathsf{CRS} \leftarrow_\$ \mathrm{CRSGen}(1^\lambda)$ *and outputs* $(\mathsf{SK}, \mathsf{PK})$*.*
- Comm *is a PPT algorithm that takes in* CRS *and set* $\{\mathsf{PK}_i\}_{i\in[n]}$ *and outputs* (K, C)*.*
- Derive *is a deterministic, polynomial time algorithm that takes in* CRS*,* SK_i*, where* $i \in [n]$*, set* $\{\mathsf{PK}_j\}_{j\in[n]}$*, and* C*, and outputs* K*.*

For correctness, we require that for any n*, and for every* $i \in [n]$*:*

$$\begin{aligned}\Pr\Big[K \leftarrow \mathrm{Derive}\Big(\mathsf{CRS}, \mathsf{SK}_i, \{\mathsf{PK}_j\}_{j\in[n]}, C\Big) :\ & (K, C) \leftarrow_\$ \mathrm{Comm}\Big(\mathsf{CRS}, \{\mathsf{PK}_j\}_{j\in[n]}\Big);\\ & \forall j \in [n], (\mathsf{SK}_j, \mathsf{PK}_j) \leftarrow_\$ \mathrm{Init}(\mathsf{CRS});\\ & \mathsf{CRS} \leftarrow_\$ \mathrm{CRSGen}(1^\lambda)\Big] = 1.\end{aligned}$$

For security, we require that for every PPT adversary $\mathcal{A}$ *that specifies* $n = \mathsf{poly}(\lambda)$*:*

$$\begin{aligned}\Pr\Big[K \leftarrow_\$ \mathcal{A}\Big(\mathsf{CRS}, \{\mathsf{PK}_i\}_{i\in[n]}, C\Big) :\ & (K, C) \leftarrow_\$ \mathrm{Comm}\Big(\mathsf{CRS}, \{\mathsf{PK}_i\}_{i\in[n]}\Big);\\ & \forall i \in [n], (\mathsf{SK}_i, \mathsf{PK}_i) \leftarrow_\$ \mathrm{Init}(\mathsf{CRS});\\ & \mathsf{CRS} \leftarrow_\$ \mathrm{CRSGen}(1^\lambda)\Big] \leq \mathsf{negl}(\lambda).\end{aligned}$$

Ideally, $|C|$ should be a small function (perhaps independent) of n.

Remark 2.5. *Of course, there is a simple CKE protocol (without CRS) from PKE scheme* $\mathrm{PKE} = (\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$*, where* $|C| = O(\lambda \cdot n)$*:* $\mathrm{Init}()$ *simply samples* $\mathsf{sk} \leftarrow_\$ \{0,1\}^\lambda$*, then computes* $\mathsf{pk} \leftarrow \mathrm{Gen}(\mathsf{sk})$ *and outputs* $(\mathsf{sk}, \mathsf{pk})$*.* $\mathrm{Comm}(\{\mathsf{pk}_i\}_{i\in[n]})$ *samples* $K \leftarrow_\$ \{0,1\}^\lambda$*, and for each* $i \in [n]$ *computes* $c_i \leftarrow_\$ \mathrm{Enc}(\mathsf{pk}_i, K)$*. It then outputs* (K, C)*, where* $C = (c_1, \ldots, c_n)$*. Finally,* $\mathrm{Derive}(\mathsf{sk}_i, \{\mathsf{pk}_j\}_{j\in[n]}, C)$ *computes* $K \leftarrow \mathrm{Dec}(\mathsf{sk}_i, c_i)$ *and outputs* K*. Correctness and security follow trivially.*

3 From CGKA to CKE Tightly

In this section, we show that CKE is at the core of CGKA, both in terms of cryptographic strength and *worst-case* communication complexity, by providing a *tight* construction of the former from the latter. The simpler counter direction—building CGKA from CKE, tightly—is provided in Appendix E. From these two reductions, we show that the worst-case communication complexity of CGKA operations is asymptotically equivalent to the size of CKE ciphertexts. That is, we show that the best possible size of a CKE ciphertext implies 1. a lower bound on the worst-case communication complexity of CGKA operations; and 2. an upper bound for the same. With this result, we additionally prove that the communication overhead in a CGKA group is necessarily increased if group members remain offline after they were added to the group. Indeed, based on our $\Omega(n)$ lower bound on CKE ciphertext size for protocols that make black-box use of PKE from Section 4, we show that worst-case communication overhead for CGKA protocols that make black-box use of PKE is $\Omega(k)$, where k is the number of added users who remain inactive after being added to the group. Furthermore, we show that this holds even for (unboundedly) many consecutive operations.

To illustrate our proof idea, consider the following execution of a CGKA protocol: Let users A and B be members of an existing CGKA group. User A adds k new users to this group before user B removes A from the group and B finally conducts a state update. After A is removed and B updates his state, the group must share a key that is secure even if A is corrupted after he is removed or B was corrupted before his update, and there were no other corruptions. (Note that these corruptions of A and B must be harmless w.r.t. security because A was removed and B updated his state to recover according to PCS.) We observe that the only information received by the k new users so far were A's add-ciphertexts and B's remove- and update-ciphertexts. Since A may have been corrupted (which reveals the randomness she used for adding the k users), the add-ciphertexts may contain no confidential payload. Similarly, B might have been corrupted until he updated his state. Hence, B's ciphertext that updates his state is the only input from which the k users can derive a secure group key. This update ciphertext intuitively corresponds to a CKE ciphertext that establishes a key with the k newly added users. In our proof, we generalize this intuition to show that, as long as k new group members remain passive, a recurring linear communication overhead in $\Omega(k)$ cannot be avoided when active group members repeatedly update the group's key material.

3.1 Embedding CGKA Ciphertexts in CKE Ciphertexts

With our proof that CGKA implies CKE, we directly lift the communication-cost lower bound for CKE from Section 4 to certain *bad* sequences in a CGKA execution. That means, our proof implies that such bad sequences in a CGKA execution lead to a linear communication overhead in the number of *affected* users. For this, we build a CKE construction that embeds specific CGKA ciphertexts in its CKE ciphertexts. Thus, a CGKA scheme that achieves sub-linear communication costs in the number of affected group members for these embedded ciphertexts results in a CKE with compact ciphertexts, which contradicts our lower bound from Section 4.

Components of Bad Sequences. Intuitively, a *bad CGKA sequence* is an operation sequence in a CGKA session during which *k passive users* are added to the group that stay offline while (few) other members actively conduct CGKA operations continuously. A CGKA session that contains such a sequence can be split into (1) a *pre-add phase* that ends when the first of these k passive users is added and (2) the subsequent *bad sequence* itself. The *bad sequence* contains (2.a) the *add operations* due to which the passive users become group members as well as (2.c) multiple, potentially overlapping, iterations of *collective update assistances*. With these *collective update assistances*, the active users update key material for the newly added passive users, which causes the communication overhead in $\Omega(k)$. From the perspective of each *collective update assistance*, the remaining operations in a *bad sequence* can be categorized into (2.b) *ineffective pre-assistance operations* and (2.d) an *irrelevant end*. (The numbering in the above enumeration reflects the order of these components within the bad sequence; We illustrate an exemplary bad sequence in Figure 2.)

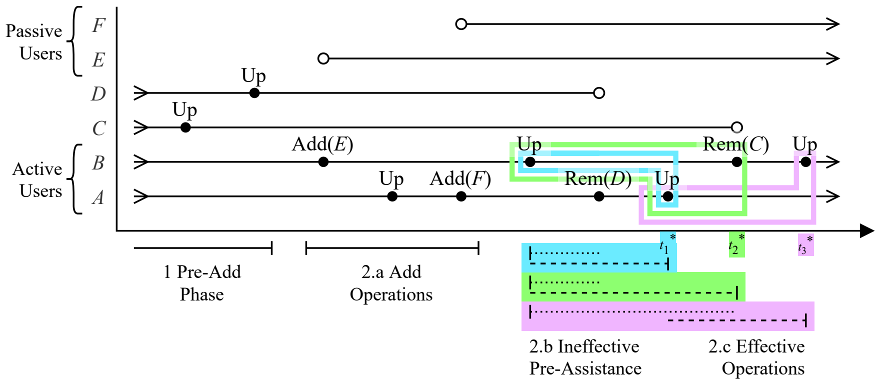

Figure 2: Bad CGKA sequence with *active users* A, B and *passive users* E, F. A and B perform three *collective update assistances* that end with operations t_1^*, t_2^*, and t_3^*, respectively. The *effective operations* of each assistance are marked with colored frames. Operations between the last *add operation* and a frame are part of the respective *ineffective pre-assistance*.

Let sequence $\mathsf{Seq} = (\mathsf{Op}_1, \dots, \mathsf{Op}_n)$ be the execution schedule of a CGKA session, where each Op_t is a tuple that refers to an executed group operation with the following format: $\mathsf{Op}_t = (\mathrm{Up}, \mathsf{PK}, \perp)$ means that PK updates their state; $\mathsf{Op}_t = (\mathrm{Add}, \mathsf{PK}, \mathsf{PK}^*)$ means that PK adds PK^*; $\mathsf{Op}_t = (\mathrm{Rem}, \mathsf{PK}, \mathsf{PK}^*)$ means that PK removes PK^*; see Section 2.1 for more details. Further, let $PU, |PU| = k$, be the public key set of the k passive users, such that for every $\mathsf{PK}^* \in PU$ there exists an operation $(\mathrm{Add}, \cdot, \mathsf{PK}^*)$ but neither an operation $(\mathrm{Rem}, \cdot, \mathsf{PK}^*)$ nor an operation $(\cdot, \mathsf{PK}^*, \cdot)$ in sequence Seq.

(1) The *pre-add phase* starts at the beginning of the entire sequence and ends with the $t_1^A - 1$ th operation, where $\mathsf{Op}_{t_1^A} = (\mathrm{Add}, \cdot, \mathsf{PK}^*)$ is the *first* operation that adds a user $\mathsf{PK}^* \in PU$ to the group. **(2.a)** The *add operations*, starting with operation $\mathsf{Op}_{t_1^A}$, end with the *last* operation $\mathsf{Op}_{t_k^A} = (\mathrm{Add}, \cdot, \mathsf{PK}^*)$ that adds a user $\mathsf{PK}^* \in PU$ to the group. (Also operations other than adding passive users can be contained in this phase.)

(2.c) The first *collective update assistance* ends when all active users conducted their first update after the add operations. During such a *collective update assistance*, the active users both propagate new *own key material* but also collectively establish and communicate new *key material for the passive users.* We define AU_{t^*} as the public key set of *users* who are *active* between the t_1^A th and t^* th operation. That means $\mathsf{PK}^* \in AU_{t^*}$ *iff* there exists at least one operation $\mathsf{Op}_t = (\cdot, \mathsf{PK}^*, \cdot)$ but no operation $\mathsf{Op}_t = (\mathrm{Rem}, \cdot, \mathsf{PK}^*)$ for $t_1^A \leq t \leq t^*$ in sequence Seq. Every *collective update assistance* by active users in set AU_{t^*} is determined by its final operation $\mathsf{Op}_{t^*}, t^* > t_k^A$, for which it must hold that all users $\mathsf{PK}^* \in AU_{t^*}$ conducted an update operation between the $t_k^A + 1$ th and t^* th operation. Such a *collective update assistance* consists of a set of *effective operations* EO_{t^*} from sequence Seq. These *effective operations* establish key material with the passive users and, in total, have a communication overhead of $\Omega(k)$ as we will prove. **(2.b)** Operations executed prior to the t^* th operation that are not in set EO_{t^*} are called *ineffective pre-assistance operations.* **(2.d)** The remaining sequence after the t^* th operation is the *irrelevant end.* In summary, a bad sequence from the perspective of one (out of potentially many) *collective update assistances* is structured as follows: (2.a) *add operations* between the t_1^A th and t_k^A th operation, (2.b) *ineffective pre-assistance* *operations* between the $t_k^A + 1\,$th and $t^* - 1\,$th operation, (2.c) *effective operations* between the $t_k^A + 1\,$th and $t^*\,$th operation that constitute this *collective update assistance*, and (2.d) *irrelevant end* after the $t^*\,$th operation.

The *effective operations* consist of all active users' operations since their respective most recent update operation. That means, for each active user public key $\mathsf{PK} \in AU_{t^*}$, the set of *effective operations* EO_{t^*} in a *collective update assistance* contains all operations $\mathsf{Op}_{t'} = (\cdot, \mathsf{PK}, \cdot)$ that were initiated since the most recent update operation $\mathsf{Op}_{t_{\mathsf{PK}}} = (\mathrm{Up}, \mathsf{PK}, \cdot)$ by user PK, where $t_{\mathsf{PK}} \leq t' \leq t^*$ with maximal t_{PK}, respectively.

Intuition for a Bad Sequence. Active users establish secret key material for passive users in *collective update assistances.* The communication overhead in $\Omega(k)$ that is induced by such a *collective update assistance* can be distributed among all corresponding *effective operations.* That means, active users can trade the work of establishing key material and the corresponding necessary communication overhead within each *collective update assistance.* However, it is important to emphasize that operations only establish key material to passive users *effectively* if the involved active users are not corrupted at that point. Hence, from the perspective of a CGKA group key computed with the $t^*\,$th operation, prior operations only contribute effectively to its secure computation if the involved users were able to recover from a potential earlier corruption. Such a recovery from a corruption is achieved via an update operation. This is the reason why the *effective operations* are defined as each active user's last operations since their most recent state update. During and after these state updates, the active users collectively assist the passive users in securely deriving the same CGKA group key in the $t^*\,$th operation.

Based on the above terminology, we formulate our communication overhead lower bound in the following theorem:

Theorem 3.1 (CGKA Lower Bound)**.** *Let* Seq *be an execution schedule of a CGKA session during which k* passive users *are added to the group until the $t_k^A\,$th operation. Let t^* determine the last operation of any subsequent* collective update assistance *such that all* active users *in set AU_{t^*} conduct an update between the t_k^A+1 th and t^* th operation. Finally, let EO_{t^*} be the corresponding set of* effective operations *that consist of all* active users' *most recent update and subsequent operations until the t^* th operation. The total size of ciphertexts sent by operations in set EO_{t^*} is $\Omega(k)$ for every CGKA construction that makes black-box use of PKE.*

We want to note that our lower bound could be extended to more *bad sequences* with equally damaging effect on the communication overhead. For clarity and compactness, we focus on the chosen specification.

Proof Sketch. The proof of Theorem 3.1 is provided in Appendix D. In summary, this proof proceeds as follows: We build a CKE construction that internally uses a CGKA scheme to execute a CGKA execution schedule Seq. For establishing a CKE key to k public keys, this sequence Seq contains at least one *collective update assistance* for *k passive users.* The core idea of the CKE construction is that precisely the *effective operations'* CGKA ciphertexts of this *collective update assistance* in the CGKA sequence are embedded in the committed CKE ciphertext. Hence, the total ciphertext size of these *effective CGKA operations* equals the size of the CKE ciphertext. All remaining operations in the CGKA sequence (i.e., *pre-add phase*, *add operations*, and *ineffective pre-assistance operations*) are, in different shapes, encoded in the CKE common reference string CRS.

The complex but interesting idea of this construction, and hence of this proof, is the isolation of the *effective operations* from the remaining operations in the entire sequence as well as their encoding in the CKE ciphertext such that CKE functionality and security are reached. As part of the proof, we reduce the security of this CKE construction to the security of the underlying CGKA scheme. Finally, we show that a CGKA scheme that executes schedule Seq without inducing a communication overhead of $\Omega(k)$ for the *effective operations* implies a CKE construction with compact ciphertexts.

In Corollary 3.2 we formulate a simpler, more specific variant of bad sequences that is directly implied by Theorem 3.1. Consider a sequence Seq in which the active users, after adding the passive users, only conduct state update operations. Then, the *effective operations* of each *collective update assistance* in sequence Seq are simply the most recent state updates by each active user.

Corollary 3.2 (Effective Update Operations)**.** *Let* Seq *be an execution schedule of a CGKA session during which* k passive users *are added to the group until the* t_k^A *th operation. Let* t^* *determine the last operation of any subsequent* collective update assistance *such that all* active users *in set* AU_{t^*} *conduct an update between the* t_k^A+1 *th and* t^* *th operation. If all operations after the* t_k^A *th operation are state updates, then the total size of ciphertexts sent due to the most recent updates by each active user in set* AU_{t^*} *is* $\Omega(k)$ *for every CGKA construction that makes black-box use of PKE, where* $|AU_{t^*}| = |EO_{t^*}|$.

Overlapping Collective Update Assistances. We want to point out that *effective operations* of different *collective update assistances* may overlap. For example, an active user A may update their state during the sequence Seq precisely once after the passive users were added. The remaining active users B and C may repeatedly perform new updates until the end of the sequence. In this case, the *effective operations* of all *collective update assistances* in sequence Seq will include the single update operation by A and always the most recent operations of B and C since their respective latest update in this sequence. As we will show in Section 5, there exists no optimal strategy to exploit the fact that effective operations of *different* collective update assistances can *overlap*. For example, one cannot successfully predict which *single* effective operations are in *several* collective update assistances and thus make these *single* operations have large communication overhead, so that large costs are not repeated several times.

Continuous Update Assistances. We finally come back to our motivating example CGKA execution schedule. In this schedule, only one user A adds the k passive users, and another user B removes A thereafter. In order to show that adding k passive users can induce a *continuous* communication overhead, we extend this execution schedule: after adding the k passive users, l active users replace each other, one after another. More precisely, first a user A adds k users as well as a second user B, then user B removes A and adds a new user C, then C replaces user B by a new user D, and so on. Each of these active users additionally performs a state update after replacing their predecessor. The effect of this cascade of replace-update sequences is that each contained update operation constitutes a single ~~*collective*~~ *update assistance*, individually inducing a communication overhead of $\Omega(k)$.[9] As a result, the entire schedule induces a communication

[9] We strike out "collective" because each update assistance is conducted by a single active user in this execution schedule.

overhead of $\Omega(k \cdot l)$. We formally define this CGKA execution schedule in Definition 3.3 and give the corresponding Corollary 3.4.

Definition 3.3 (Continuous Update Assistance)**.** *Let* Seq *be an operation schedule of a CGKA session during which user* PK_0 *adds* k passive users *to the group until the* t_k^A *th operation. Schedule* Seq *contains a* Continuous Update Assistance *of length* l *after the* t_k^A *th operation if* Seq *proceeds after the* t_k^A *th operation with* l *repetitions of operation sequences* $(\mathsf{Op}_{i,A}, \mathsf{Op}_{i,R}, \mathsf{Op}_{i,U}), i \in [l]$*, where* $\mathsf{Op}_{i,A} = (\mathrm{Add}, \mathsf{PK}_i, \mathsf{PK}_{i+1})$*,* $\mathsf{Op}_{i,R} = (\mathrm{Rem}, \mathsf{PK}_{i+1}, \mathsf{PK}_i)$*, and* $\mathsf{Op}_{i,U} = (\mathrm{Up}, \mathsf{PK}_{i+1}, \perp)$ *for independent users* $\mathsf{PK}_j, j \in [l+1]$.

Corollary 3.4 (Continuous Communication Overhead)**.** *For every CGKA execution schedule* Seq *that contains a* Continuous Update Assistance *of length* l *after the* t_k^A *th operation, the total size of ciphertexts output by the* $3l$ *operations after the* t_k^A *th operation is* $\Omega(k \cdot l)$ *for every CGKA construction that makes black-box use of PKE.*

The proof of Corollary 3.4 is a direct application of Theorem 3.1 via a simple hybrid argument that considers each replace-update sequence in Seq as a ~~*collective*~~ *update assistance.*

4 CKE Lower Bound from PKE

Before showing our lower bound for CKE from PKE, we need to define the model in which we prove it.

Preliminaries. For a function f we write $f(*) = y$ to indicate $f(x) = y$ for some input x. We generalize this notation for the case in which some part of the input is fixed, writing $f(a_1, *) = y$, interpreted in the natural way. Many other preliminaries are deferred to Appendix A.

CKE in the Ψ-model. The model for our proof gives the protocol and adversary access to an oracle distribution, defined as follows:

Definition 4.1. *We define an oracle distribution* Ψ *that produces oracles* $(\mathbf{O}, \mathbf{u}, \mathbf{v})$*, where* $\mathbf{O} = (\mathbf{g}, \mathbf{e}, \mathbf{d})$*. The distribution is parameterized over a security parameter* λ*, but we keep it implicit for better readability.*

- $\mathbf{g}\colon \{0,1\}^\lambda \mapsto \{0,1\}^{3\lambda}$ *is a random length-tripling function, mapping a secret key to a public key.*
- $\mathbf{e}\colon \{0,1\}^{3\lambda} \times \{0,1\} \times \{0,1\}^\lambda \mapsto \{0,1\}^{3\lambda}$*: is a random function satisfying the following: for every* $\mathsf{pk} \in \{0,1\}^{3\lambda}$*, the function* $\mathbf{e}(\mathsf{pk}, \cdot, \cdot)$ *is injective; i.e., if* $(m, r) \neq (m', r')$*, then* $\mathbf{e}(\mathsf{pk}, m, r) \neq \mathbf{e}(\mathsf{pk}, m', r')$.
- $\mathbf{d} : \{0,1\}^\lambda \times \{0,1\}^{3\lambda} \mapsto \{0,1\}$ *is the decryption oracle, where* $\mathbf{d}(\mathsf{sk}, c)$ *outputs* $m \in \{0,1\}$ *if* $\mathbf{e}(\mathbf{g}(\mathsf{sk}), m, *) = c$*; otherwise,* $\mathbf{d}(\mathsf{sk}, c) = \perp$.
- $\mathbf{v}\colon \{0,1\}^{3\lambda} \times \{0,1\}^{3\lambda} \mapsto \{\perp, \top\}$*, is a ciphertext-validity checking oracle:* $\mathbf{v}(\mathsf{pk}, c)$ *outputs* $\top$ *if* c *is in the range of* $\mathbf{e}(\mathsf{pk}, \cdot, \cdot)$ *(that is,* $c := \mathbf{e}(\mathsf{pk}, *, *)$*); otherwise, it outputs* $\perp$.

- $\mathbf{u}\colon \{0,1\}^{3\lambda} \times \{0,1\}^{3\lambda} \mapsto \{0,1\} \cup \{\bot\}$, *is an oracle that decrypts wrt invalid public keys; given* (pk, c)*, if there exists* sk *such that* $\mathbf{g}(sk) = \mathsf{pk}$*, then* $\mathbf{u}(\mathsf{pk}, c) = \bot$*; otherwise, if there exists a message* $m \in \{0,1\}$ *such that* $\mathbf{e}(\mathsf{pk}, m, *) = c$*, return* m*; else, return* $\bot$*.*

Now, we can define CKE in the Ψ-model.

Definition 4.2. *A* Compact Key Exchange *scheme in the* Ψ*-model is defined equivalently as in Definition 2.4, except that each of the CKE algorithms and the adversary additionally have access to the* Ψ *oracles. We denote such access using* Ψ *as a superscript in the corresponding algorithms, e.g.,* $\mathrm{Init}^{\Psi}(\mathsf{CRS})$*. All other syntax and security requirements stay the same.*

4.1 Proof Outline

Our lower bound is derived from the following two lemmas. The first lemma shows a random $(\mathbf{g}, \mathbf{e}, \mathbf{d})$ constitutes an ideally-secure PKE protocol, even against adversaries that have access to the oracles $(\mathbf{u}, \mathbf{v})$, in addition to $(\mathbf{g}, \mathbf{e}, \mathbf{d})$. The second lemma shows that the security of any proposed CKE protocol (CRSGen, Init, Comm, Derive), instantiated with a random $\mathbf{O} := (\mathbf{g}, \mathbf{e}, \mathbf{d})$, may be broken by an adversary making at most a polynomial number of queries to $(\mathbf{O}, \mathbf{u}, \mathbf{v})$. The black-box separation will then follow.

Lemma 4.3 **($\mathbf{O}$ is secure against $(\mathbf{O}, \mathbf{u}, \mathbf{v})$).** *For any polynomial-query adversary* A*:* $\Pr[\mathsf{A}^{\mathbf{O},\mathbf{u},\mathbf{v}}(\mathsf{pk}, c) = b] \le 1/2 + \frac{1}{2^{\lambda/2}}$*, where* $(\mathbf{g}, \mathbf{e}, \mathbf{d}, \mathbf{u}, \mathbf{v}) \leftarrow_{\$} \Psi$*,* $\mathbf{O} := (\mathbf{g}, \mathbf{e}, \mathbf{d})$*,* $b \leftarrow_{\$} \{0,1\}$*,* $\mathsf{sk} \leftarrow_{\$} \{0,1\}^{\lambda}$*,* $\mathsf{pk} = \mathbf{g}(\mathsf{sk})$*,* $r \leftarrow_{\$} \{0,1\}^{\lambda}$ *and* $c = \mathbf{e}(\mathsf{pk}, b; r)$*.*

The following lemma shows how to break compact CKE constructions relative to the PKE oracles. The lemma shows that even for encrypting single-bit keys (i.e., $|K| = 1$), a CKE ciphertext cannot be sub-linear in n.

Lemma 4.4 (Breaking CKE relative to $(\mathbf{O}, \mathbf{u}, \mathbf{v})$)**.** *Let* (CRSGen*,* Init*,* Comm*,* Derive) *be a candidate black-box construction of CKE, where for any CKE ciphertext* C*,* $|C| \le \frac{3\lambda(n-1)}{2}$*. For any constant* c*, there exists a polynomial-query adversary* $\mathsf{Brk}^{\mathbf{O},\mathbf{u},\mathbf{v}}$ *such that* $\Pr[\mathsf{Brk}^{\mathbf{O},\mathbf{u},\mathbf{v}}(\mathsf{PK}_1, \ldots, \mathsf{PK}_n, C) = K] \ge 1 - \frac{1}{\lambda^c}$*, where* $(\mathbf{g}, \mathbf{e}, \mathbf{d}, \mathbf{u}, \mathbf{v}) \leftarrow_{\$} \Psi$*,* $\mathbf{O} := (\mathbf{g}, \mathbf{e}, \mathbf{d})$*,* $\mathsf{CRS} \leftarrow_{\$} \mathrm{CRSGen}^{\mathbf{O}}(1^{\lambda})$*,* $(\mathsf{PK}_i, *) \leftarrow_{\$} \mathrm{Init}^{\mathbf{O}}(\mathsf{CRS})$ *for* $i \in [n]$*, and* $(K, C) \leftarrow_{\$} \mathrm{Comm}^{\mathbf{O}}(\mathsf{CRS}, \mathsf{PK}_1, \ldots, \mathsf{PK}_n)$*.*

Roadmap. Lemma 4.3 is proved in a straightforward way (hence omitted), given the random nature of the oracles. The proof of Lemma 4.4 is the main technical bulk of our paper, consisting of the description of an attacker and attack analysis. We first describe the attacker for the case $(\mathrm{Init}^{\mathbf{g}}, \mathrm{Comm}^{\mathbf{e}}, \mathrm{Derive}^{\mathbf{d}})$ in Section 4.2, and will then describe an attack against general constructions in Appendix C. Lemma 4.4 will follow similarly from the below simpler attack. We may now obtain the following from Lemmas 4.3,4.4, proved via standard black-box separation techniques.

Theorem 4.5. *There exists no fully-black-box construction of CKE schemes from PKE schemes with CKE ciphertext size* $o(n)|c|$*, where* $|c|$ *denotes the ciphertext size of the base PKE scheme.*

4.2 Attack for $(\mathrm{CRSGen}^{\mathbf{g}}, \mathrm{Init}^{\mathbf{g}}, \mathrm{Comm}^{\mathbf{e}}, \mathrm{Derive}^{\mathbf{d}})$

We will show an attack for the case in which oracle access is of the form $(\mathrm{CRSGen}^{\mathbf{g}}, \mathrm{Init}^{\mathbf{g}}, \mathrm{Comm}^{\mathbf{e}}, \mathrm{Derive}^{\mathbf{d}})$. This already captures the main ideas behind the impossibility result. We will then show how to relax this assumption.

Attack overview. Let $(\mathsf{PK}_1, \ldots, \mathsf{PK}_n, C)$ be the public keys and the ciphertext. We show an impossibility as long as $|C| \leq \frac{3\lambda(n-1)}{2}$, where recall that 3λ is the size of a base ciphertext as per oracles generated by Ψ (Definition 4.1). This particular choice for the size of C will ensure that C can "embed" at most $n-1$ base ciphertexts, in a sense we will later define in Lemma A.2.

For simplicity, in this overview we assume that the scheme does not have a CRS. The attack is based on the following high-level idea. During the generation of each $(\mathsf{PK}_i, \mathsf{SK}_i) \leftarrow_\$ \mathrm{Init}^{\mathbf{g}}(1^\lambda)$ a set of $\mathbf{g}$-type query/answer pairs made. Let $\mathsf{KPair}_i = \{(\mathsf{pk}_{i,1}, \mathsf{sk}_{i,1}), \ldots, (\mathsf{pk}_{i,t}, \mathsf{sk}_{i,t})\}$ be the set of public/secret key pairs produced during the generation of PK_i. These public keys are in someway encoded in PK_i, and the ability to decrypt with respect to these base $\mathsf{pk}_{i,j}$ public keys is the only advantage that the ith party, who has SK_i, has over an adversary.

Consider a random execution of $(K, C) \leftarrow_\$ \mathrm{Comm}^{\mathbf{e}}(\mathsf{PK}_1, \ldots, \mathsf{PK}_n)$, and let $\mathsf{Q} = \{(\mathsf{pk}_1, b_i, r_i, c_i) \mid i \in [f]\}$ contain the set of all query/answer pairs, and let $\mathsf{Q}_c = \{c_1, \ldots, c_f\}$. Since the ciphertext C is compact, C can embed at most $(n-1)$ ciphertexts c_i from the set Q_c. By embedding we mean anyone, including the legitimate users, given only C can extract at most $n-1$ valid pairs (pk_i, c_i) without querying $\mathbf{e}$.

Now for each user consider its local decryption execution. Each user performing decryption will need to decrypt pairs of the form (pk, c), in order to recover a shared K. We focus on those pairs which are valid, meaning that c is in the range of $\mathbf{e}(\mathsf{pk}, \cdot, \cdot)$. Looking ahead, the reason for this is that for invalid pairs for which the answer is $\perp$, an adversary can already simulate the answer by calling $\mathbf{u}$. Let S'_i be the set of valid pairs that come up during decryption performed by user i. Since C embeds at most $n-1$ valid pairs (pk, c), for some user h: $\mathsf{S}'_h \subseteq \mathsf{S}'_1 \cup \ldots \mathsf{S}'_{h-1}$. In other words, the set of base trapdoors needed to decrypt S'_h is a subset of those for $\mathsf{S}'_1 \cup \ldots \mathsf{S}'_{h-1}$. Moreover, in order for any user to be able to decrypt some (pk, c), the user should have observed a query/answer pair $(\mathsf{pk}, \mathsf{sk})$ during its execution of $\mathrm{Init}^{\mathbf{g}}(1^\lambda)$. Thus, recalling KPair_h, the set of base secret keys needed to decrypt elements in S'_h is a subset of $\mathsf{KPair}_1 \cup \ldots, \cup \mathsf{KPair}_{h-1}$. But each of these KPair_i sets (for $i \in [n]$) is obtained by running $\mathrm{Init}^{\mathbf{g}}(1^\lambda)$ on a security parameter, and so if an adversary runs $\mathrm{Init}^{\mathbf{g}}(1^\lambda)$ many times and collects all query/answer pairs in a set Freq, the adversary with high probability will collect all the trapdoors needed to successfully decrypt for at least one user.

How to perform simulated decryption? So far, the discussion above says that an adversary can collect a set Freq which with high probability contains all $(\mathsf{pk}, \mathsf{sk})$ pairs needed to decrypt with respect to at least one user. But even given Freq, it is unclear how to perform decryption for any user. The adversary cannot simply "look at" Freq and somehow decrypt C — the adversary will need a secret key SK to be able to run $\mathrm{Derive}(\mathsf{SK}, \cdot)$. The solution is to let the adversary sample a "fake" secret keys for users, in a manner consistent with query-answer knowledge of Freq.

We make the following assumption for the construction ($\mathrm{CRSGen}^{\mathbf{g}}$, $\mathrm{Init}^{\mathbf{g}}$, $\mathrm{Comm}^{\mathbf{e}}$, $\mathrm{Derive}^{\mathbf{d}}$) that we want to prove an impossibility for. The assumption is made only for ease of exposition.

Assumption 4.6. *We assume for any oracle* $(\mathbf{g}, \mathbf{e}, \mathbf{d}) \leftarrow_\$ \Psi$ *picked as in Definition 4.1, each algorithm in* ($\mathrm{CRSGen}^{\mathbf{g}}, \mathrm{Init}^{\mathbf{g}}$, $\mathrm{Comm}^{\mathbf{e}}$, $\mathrm{Derive}^{\mathbf{d}}$) *makes only a security parameter* λ *number of queries.*

Definition 4.7 (Partial oracles and consistency)**.** *We say a partial oracle* O_1 *(defined only on a subset of all points) is* Ψ*-valid if for some* $O_2 \in \mathsf{Supp}(\Psi)$*:* $O_1 \subseteq O_2$*, where* Supp *denotes the support of a distribution. We say an oracle* $(\mathbf{g}, \mathbf{e}, \mathbf{d})$ *is PKE-valid if it satisfies PKE completeness. A partial PKE-valid oracle is one which is a subset of a PKE-valid oracle. Note that any* Ψ*-valid oracle is*

PKE-valid as well. We say a partial oracle O_1 is consistent with a set of query/response pairs S *if* $O_1 \cup \mathsf{S}$ *is PKE-valid.*

We also need to define the notion of a partial oracle forbidding a set of query/response pairs. This technique of forbidding a set of query/answer pairs will be used extensively in our constructions, and to the best of our knowledge, no previous impossibility results deal with this technique.

Definition 4.8 (Forbidding queries)**.** *Let* Forbid *consists of "wildcard" queries/ responses, of the form* $(q \underset{z}{\rightarrow} *)$ *or* $(* \underset{z}{\rightarrow} u)$*, where* $z \in \{\mathbf{g}, \mathbf{e}\}$*. We say that a partial oracle* $O_1 = (\tilde{\mathbf{g}}, \tilde{\mathbf{e}})$ *forbids* Forbid *if (a) for any* $(q \underset{z}{\rightarrow} *) \in \mathsf{Forbid}$ *the oracle* $\tilde{z}$ *is not defined on input* q *and (b) for any* $(* \underset{z}{\rightarrow} u)$ *the oracle* $\tilde{z}$ *is not defined on any input point with a corresponding output* u *(i.e.,* y *is not in the set of output points defined under* $\tilde{z}$*).*

The attacker will first perform many random executions of $\mathrm{Init}^{\mathbf{g}}(\mathsf{CRS})$ to collect all likely query/response pairs: those that appear during a random execution with a high-enough probability. This will allow the adversary to learn the secret keys for all likely base pk's that might be embedded to more than one user's CKE public key. Once this step is done, the attacker will sample partial oracles that are consistent with the set of collected query/answer pairs. Recall that by Assumption 4.6 any execution of $\mathrm{Init}^{\mathbf{g}}(\mathsf{CRS})$ makes exactly λ queries. We say a partial oracle $\mathbf{O}'$ (defined only on a subset of points) is minimal for an execution $\mathrm{Init}^{\mathbf{O}'}(\mathsf{CRS}; R)$, if the execution makes queries only to those points defined in $\mathbf{O}'$, and nothing else. This means in particular that $\mathbf{O}'$ is defined only on λ points. In the definition below, we talk about sampling *minimal* partial oracles $\mathbf{O}'$ that agree with some set of query/answer pairs.

Definition 4.9 (Sampling partial oracles)**.** *We define the procedure* $\mathsf{ConsOrc}$*. In this definition we assume that the algorithm* $\mathrm{Init}^{\mathbf{g},\mathbf{e}}$ *makes both* $\mathbf{g}$ *and* $\mathbf{e}$ *queries (as opposed to* $\mathbf{g}$ *only), since this definition will also be used for the general attack.*

- ***Input:*** $(\mathsf{CRS}, \mathsf{PK}, \mathsf{Freq}, \mathsf{Forbid})$*: A CRS* CRS*, public key* PK*, and set of query/answer pairs* Freq *and a set of query/answer pairs* Forbid*. The set* Forbid *consists of "wildcard" forbidden queries/responses, of the form* $(q \underset{z}{\rightarrow} *)$ *or* $(* \underset{z}{\rightarrow} u)$*, where* $z \in \{\mathbf{g}, \mathbf{e}\}$*.*

- ***Output:*** $(\mathsf{SK}, \mathbf{O}')$ *or* $\perp$*, produced as follows. Sample a partial* Ψ*-generated* $\mathbf{O}' = (\mathbf{g}', \mathbf{e}')$ *defined only on* λ *queries (see Assumption 4.6), sample randomness* R *and a resultant* SK *uniformly at random subject to the conditions that (a)* $\mathbf{O}'$ *is consistent with* Freq*; (b)* $\mathbf{O}'$ *forbids* Forbid *(Definition 4.8) (c)* $\mathrm{Init}^{\mathbf{O}'}(\mathsf{CRS}; R) = (\mathsf{PK}, \mathsf{SK})$ *and (d)* $\mathbf{O}'$ *is R-minimal: the execution of* $\mathrm{Init}^{\mathbf{O}'}(\mathsf{CRS}; R)$ *makes only queries to those in* $\mathbf{O}'$*, and nothing else. If no such* $(\mathsf{SK}, \mathbf{O}')$ *exists, output* $\perp$*.*[10]

In our attack, the adversary will try performing simulated decryptions for different parties. The adversary will do so by sampling a simulated secret key $\widetilde{\mathsf{SK}}$ for that party, along with a partial oracle $\mathbf{g}'$ relative to which $\widetilde{\mathsf{SK}}$ is a secret key for that party's public key PK (i.e., $(\mathsf{PK}, \widetilde{\mathsf{SK}}) \leftarrow_{\$} \mathrm{Init}^{\mathbf{g}'}(\mathsf{CRS})$). The adversary will then perform decryption with respect to an oracle $\mathbf{g}' \Diamond^* \mathbf{O}$ that is the result of superimposing $\mathbf{g}'$ on the real oracle $\mathbf{O}$. We will define the superimposed oracle below. Essentially, the superimposed oracle is defined in a way so that it agrees with $\mathbf{g}'$, it is a valid PKE oracle, and also agrees with the real oracle as much as possible. In the definition below we define this

[10] This can happen because of the presence of forbidding queries in Forbid.

superimposing process, but note that we are not claiming that the output of $\mathbf{g}'\Diamond^*\mathbf{O}$ on a given query can be necessarily obtained by making a polynomial number of queries to $\mathbf{O}$.

As notation we use $(\mathsf{sk}_1 \underset{\mathbf{g}}{\to} \mathsf{pk}_1)$ to denote a query/answer pair of $\mathbf{g}$-type. We use similar notation for other types of queries. If L is a set of query/answer pairs, we use $\mathsf{Query}(\mathsf{L})$ to denote the query parts of the elements of L.

Definition 4.10 (Composed Oracles $\Diamond^*$)**.** *Let $\mathbf{O} := (\mathbf{g}, \mathbf{e}, \mathbf{d})$ be a Ψ-valid oracle (a possible output of Ψ) and let*

$$\mathbf{g}' := \{(\mathsf{sk}_1 \underset{\mathbf{g}}{\to} \mathsf{pk}_1), \ldots, (\mathsf{sk}_w \underset{\mathbf{g}}{\to} \mathsf{pk}_w)\}$$

be a partial Ψ-valid oracle consisting of only $\mathbf{g}$-type queries. We define a composed oracle $\mathbf{g}'\Diamond^*\mathbf{O} := (\widetilde{\mathbf{g}}, \mathbf{e}, \widetilde{\mathbf{d}})$ as follows.

- $\widetilde{\mathbf{g}}(\cdot)$*: for a given sk, let $\widetilde{\mathbf{g}}(\mathsf{sk}) \triangleq \mathsf{pk}_i$ if $\mathsf{sk} = \mathsf{sk}_i$ for $i \in [w]$; otherwise, $\widetilde{\mathbf{g}}(\mathsf{sk}) \triangleq \mathbf{g}(\mathsf{sk})$.*
- $\widetilde{\mathbf{d}}(\cdot, \cdot)$*: for a given pair (sk, c), define $\widetilde{\mathbf{d}}(\mathsf{sk}, c)$ as follows. Assuming $\mathsf{pk} = \widetilde{\mathbf{g}}(\mathsf{sk})$, if there exists $m \in \{0, 1\}$ such that $c = \mathbf{e}(\mathsf{pk}, m, *)$, return m; otherwise, return $\perp$.*

In the definition above notice that the resulting oracle $(\widetilde{\mathbf{g}}, \mathbf{e}, \widetilde{\mathbf{d}})$ is Ψ-valid (i.e., and hence a valid PKE oracle, satisfying PKE completeness) as long as $\mathbf{O}$ and $\mathbf{g}'$ are Ψ-valid. Thus, we have the following lemma.

Lemma 4.11. *Assuming $\mathbf{O}$ and g' are Ψ-valid, $(\widetilde{\mathbf{g}}, \mathbf{e}, \widetilde{\mathbf{d}})$, obtained as in Definition 4.10, is Ψ-valid, and hence PKE-valid.*

4.2.1 Description of the Attacker Against $(\mathrm{CRSGen}^{\mathbf{g}}, \mathrm{Init}^{\mathbf{g}}, \mathrm{Comm}^{\mathbf{e}}, \mathrm{Derive}^{\mathbf{d}})$

$\mathsf{Brk}^{\mathbf{g},\mathbf{e},\mathbf{d},\mathbf{u},\mathbf{v}}(\mathsf{CRS}, \mathsf{PK}_1, \ldots, \mathsf{PK}_n, C)$: The attack is based on two integers η, η', instantiated later.

1. Let $\mathsf{Decrypt} = 0$, $\mathsf{Forge} = 0$, and $\mathsf{Forbid} = \emptyset$.
2. Do the following for η iterations. Sample randomness R and execute $\mathrm{Init}^{\mathbf{g}}(\mathsf{CRS}; R)$ and record all query/response pairs in Freq.
3. Sample $\gamma \leftarrow_{\$} [\eta']$ and do the following γ times. Run $(\mathsf{PK}, *) \leftarrow_{\$} \mathrm{Init}^{\mathbf{g}}(\mathsf{CRS})$ using fresh randomness, sample $(\widetilde{\mathsf{SK}}, \mathbf{g}') \leftarrow_{\$} \mathsf{ConsOrc}(\mathsf{CRS}, \mathsf{PK}, \mathsf{Freq}, \mathsf{Forbid})$, and for every $(\mathsf{sk} \underset{\mathbf{g}}{\to} \mathsf{pk}) \in \mathbf{g}' \setminus \mathsf{Freq}$, add $(\mathsf{sk} \underset{\mathbf{g}}{\to} *)$ to Forbid; also, if $(* \underset{\mathbf{g}}{\to} \mathsf{pk}) \notin \mathsf{Freq}$, add $(* \underset{\mathbf{g}}{\to} \mathsf{pk})$ to Forbid.[11] At the end of each iteration, update Freq by adding all queries made during $(\mathsf{PK}, *) \leftarrow_{\$} \mathrm{Init}^{\mathbf{g}}(\mathsf{CRS})$ to Freq.
4. For $i \in [n]$
   (a) Sample $(\widetilde{\mathsf{SK}}_i, \mathbf{g}'_i) \leftarrow_{\$} \mathsf{ConsOrc}(\mathsf{CRS}, \mathsf{PK}_i, \mathsf{Freq}, \mathsf{Forbid})$. If $(\widetilde{\mathsf{SK}}_i, \mathbf{g}'_i) = \perp$, then halt.[12]
   (b) Let $\mathbf{g}'_i\Diamond^*\mathbf{O} = (\widetilde{\mathbf{g}}, \mathbf{e}, \widetilde{\mathbf{d}})$.

[11] Note that since Init makes only $\mathbf{g}$ queries, the output of $\mathsf{ConsOrc}$ (Definition 4.9) does not have an $\mathbf{e}'$ oracle.

[12] This can happen because the set Forbid makes some queries/responses off-limits.

(c) Execute $\mathrm{Derive}^{\widetilde{\mathbf{d}}}(\widetilde{\mathsf{SK}}_i, \{\mathsf{PK}_i\}, C)$ and answer the queries as follows. For a query $\mathsf{qu} := ((\mathsf{sk}, c) \underset{\widetilde{\mathbf{d}}}{\to} ?)$, if $(\mathsf{sk} \underset{\mathbf{g}}{\to} *) \in \mathsf{Freq}$ or $(\mathsf{sk} \underset{\mathbf{g}}{\to} *) \notin \mathbf{g}'$, then reply to the query with $\mathbf{d}(\mathsf{sk}, c)$. Otherwise, letting $\mathsf{pk} = \widetilde{\mathbf{g}}(\mathsf{sk})$ — which can be computed efficiently — if $(\mathsf{sk}' \underset{\mathbf{g}}{\to} \mathsf{pk}) \in \mathsf{Freq}$ for some sk', reply to qu with $\mathbf{d}(\mathsf{sk}', c)$. Else,

   i. if $\mathbf{v}(\mathsf{pk}, c) = \perp$, then reply to qu with $\perp$;

   ii. else if $\mathbf{u}(\mathsf{pk}, c) = m \neq \perp$, then reply to qu with m;

   iii. else, add (pk, c) to Chal, and if $i < n$, go to the next i (Step 4); otherwise, set $\mathsf{Forge} = 1$ and halt.

(d) If we have not halted so far, letting $\widetilde{K_i}$ be the output of the simulated decryption $\mathrm{Derive}^{\widetilde{\mathbf{d}}}(\mathsf{CRS}, \mathsf{PK}_1, \ldots, \mathsf{PK}_n, \widetilde{\mathsf{SK}}_i, C)$, return $\widetilde{K_i}$, set $\mathsf{Decrypt} = 1$ and halt.

Notation 4.12. *Let* $(\mathsf{CRS}, \mathsf{PK}_1, \ldots, \mathsf{PK}_n, C)$ *be as above. Let* QC *be the set of query/response pairs made to generate* $\mathsf{CRS} \leftarrow_{\$} \mathrm{CRSGen}^{\mathbf{g}}(1^{\lambda})$*. For* $i \in [n]$ *let* QGen_i *be the set of query/response pairs made to generate* $(\mathsf{PK}_i, \mathsf{SK}_i) \leftarrow_{\$} \mathrm{Init}^{\mathbf{g}}(\mathsf{CRS})$*. Let* K *be the corresponding key for* C*, and suppose* QEnc *is the set of query/response pairs made to generate* $(K, C) \leftarrow_{\$} \mathrm{Comm}^{\mathbf{e}}(\mathsf{CRS}, \mathsf{PK}_1, \ldots, \mathsf{PK}_n)$*.*

4.2.2 Attack Analysis

We define some events that will help us to analyze the effectiveness of the attack.

Definition 4.13. *We define the following events, based on the variables introduced in Notation 4.12 and for attacker* Brk*.*

- *Event* Evnt_1 *is the event that* $\mathsf{Decrypt} = 1$*, and* Evnt_2 *is the event that* $\mathsf{Forge} = 1$*.*
- *Event* Empty_i *for* $i \in [n]$*: the event that* $\mathbf{g}'_i = \emptyset$*.*[13] *We let* $\mathsf{Empty} := \bigvee_i \mathsf{Empty}_i$*.*
- *Event* Agree*: for all* $h \in [n]$*,* $\mathbf{g}'_h$ *agrees with* $\cup_{i \neq h} \mathsf{QGen}_i \cup \mathsf{QC} \cup \mathsf{QEnc}$*. If* g'_h *is empty, then agreement is assumed to hold.*
- *Event* $\mathsf{Surprise}_i$ *for* $i \in [n]$*: there exists* $(* \underset{\mathbf{g}}{\to} \mathsf{pk}) \in \mathbf{g}'_i$ *such that* $(* \underset{\mathbf{g}}{\to} \mathsf{pk}) \in \mathsf{QC}$ *and* $(* \underset{\mathbf{g}}{\to} \mathsf{pk}) \notin \mathsf{Freq}$*. If* $\mathbf{g}'_i$ *is empty, we say* $\mathsf{Surprise}_i$ *does not hold. We let* $\mathsf{Surprise} := \bigvee_i \mathsf{Surprise}_i$*.*
- *Event* Spoof*: the event that for some* $h \in [n]$*, there exists* $(* \underset{\mathbf{g}}{\to} \mathsf{pk}) \in \mathbf{g}'_h$ *such that (a)* $(* \underset{\mathbf{g}}{\to} \mathsf{pk}) \notin \cup_{i \in [n]} \mathsf{QGen}_i \cup \mathsf{QC} \cup \mathsf{Freq}$ *and (b)* pk *is* $\mathbf{g}$*-valid; namely,* $\mathbf{g}(*) = \mathsf{pk}$*.*
- *Event* $\mathsf{Intersect}$*: the event that for two distinct* $i, j \in [n]$*, there is either an intersection query between* QGen_i *and* QGen_j *not picked up by* Freq*, or there is an intersection response between* QGen_i *and* QGen_j *not picked up by* Freq*. That is, the event that* $(\mathsf{QGen}_i \cap \mathsf{QGen}_j) \setminus \mathsf{Freq} \neq \emptyset$ *or there exists* pk *such that* $(* \underset{\mathbf{g}}{\to} \mathsf{pk}) \in \mathsf{QGen}_i$ *and* $(* \underset{\mathbf{g}}{\to} \mathsf{pk}) \in \mathsf{QGen}_j$ *and* $(* \underset{\mathbf{g}}{\to} \mathsf{pk}) \notin \mathsf{Freq}$*.*

In the following lemmas we will bound the probability of each of the above events. Lemma 4.20 will make use of these bounds to bound the probability of the attack being successful.

[13] This is the same thing as $(\widetilde{\mathsf{SK}}_i, \mathbf{g}'_i) = \perp$, namely the output of $\mathsf{ConsOrc}()$, in Definition 4.9, is $\perp$. The output of $\mathsf{ConsOrc}()$ may indeed be $\perp$ due to the presence of forbidding queries in Forbid.

Lemma 4.14. *Assuming $\eta \geq \lambda^{0.1}$, for any $i \in [n]$* $\Pr[\mathsf{Empty}_i] \leq \frac{1}{2^{\omega(\log\lambda)}} + \frac{2\lambda^{1.1}\eta'}{\eta}$*. Thus,* $\Pr[\mathsf{Empty}] \leq \frac{n}{2^{\omega(\log\lambda)}} + \frac{2n\lambda^{1.1}\eta'}{\eta}$

Lemma 4.15. *Assuming $\eta \geq \lambda^{0.1}$,* $\Pr[\mathsf{Agree}] \geq 1 - \frac{n}{2^{\omega(\log(\lambda))}} - \frac{n\lambda}{\eta'} - \frac{n^2\lambda^{1.1}}{\eta}$*.*

Lemma 4.16. *For any $i \in [n]$* $\Pr[\mathsf{Surprise}_i] \leq \frac{\lambda}{\eta'}$*. As a result,* $\Pr[\mathsf{Surprise}] \leq \frac{n\lambda}{\eta'}$*.*

Lemma 4.17. *We have* $\Pr[\mathsf{Spoof}] \leq \frac{1}{2^{2\lambda}}$*.*

Lemma 4.18. *Assuming $\eta \geq \lambda^{0.1}$,* $\Pr[\mathsf{Intersect}] \leq \frac{2n^2\lambda^{1.1}}{\eta} + \frac{n^2}{2^{\omega(\log\lambda)}}$*.*

Proof. Let $p = \frac{\lambda^{0.1}}{\eta}$ and note that $p\eta \geq \omega(\log\lambda)$. By Lemma A.7, for any fixed and distinct i and j, the probability that $(\mathsf{QGen}_i \cap \mathsf{QGen}_j) \setminus \mathsf{Freq} \neq \emptyset$ is at most $\frac{2\lambda^{1.1}}{\eta} + \frac{1}{2^{\omega(\log\lambda)}}$. The proof now follows by the union bound over all pairs of (i, j), which is less than n^2. □

Lemma 4.19. *Suppose $|C| \leq \frac{3\lambda(n-1)}{2}$, where C is the CKE ciphertext. For any constant $c > 0$, assuming $\eta' \geq n\lambda^{c+1}$ and $\eta \geq n\eta'\lambda^{1.1+c}$,* $\Pr[\mathsf{Evnt}_2] \leq \frac{5}{\lambda^c}$*.*

Lemma 4.20 (Attack effectiveness)**.** *Suppose $|C| \leq \frac{3\lambda(n-1)}{2}$, where C is the CKE ciphertext. For any constant $c > 0$, assuming $\eta' \geq n\lambda^{c+1}$ and $\eta \geq 4n\eta'\lambda^{1.1+c}$,* $\Pr[\widetilde{K} = K] \geq 1 - \frac{10}{\lambda^c}$*.*

Proof. We have $\Pr[\mathsf{Evnt}_1 \vee \mathsf{Evnt}_2 \vee \mathsf{Empty}] = 1$. This is because whenever Brk halts, the halting condition specified in Line 4a (event Empty), or Line 4(c)iii (event Evnt_2) or Line 4d (event Evnt_1) must be triggered. Thus, $\Pr[\overline{\mathsf{Evnt}_1}] \leq \Pr[\mathsf{Evnt}_2] + \Pr[\mathsf{Empty}]$. By Lemma 4.15 $\Pr[\overline{\mathsf{Agree}}] \leq -\frac{n}{2^{\omega(\log(\lambda))}} - \frac{n\lambda}{\eta'} - \frac{n^2\lambda^{1.1}}{\eta}$. By the particular values of η and η', $\Pr[\overline{\mathsf{Agree}}] \leq \frac{3}{\lambda^c}$. Moreover, by invoking Lemmas 4.14 and 4.19 for these values of η and η', we have $\Pr[\mathsf{Empty}] \leq \frac{1}{\lambda^c}$ $\Pr[\mathsf{Evnt}_2] \leq \frac{5}{\lambda^c}$.

$$\begin{aligned}\Pr[\widetilde{K} = K] &\geq \Pr[\widetilde{K} = K \mid \mathsf{Evnt}_1 \wedge \mathsf{Agree} \wedge \overline{\mathsf{Empty}}] \Pr[\mathsf{Evnt}_1 \wedge \mathsf{Agree} \wedge \overline{\mathsf{Empty}}] \\ &\geq 1(1 - \Pr[\mathsf{Evnt}_2] - \Pr[\mathsf{Empty}] - \Pr[\overline{\mathsf{Agree}}] - \Pr[\mathsf{Empty}]) \geq 1 - \frac{5}{\lambda^c} - \frac{3}{\lambda^c} - \frac{2}{\lambda^c} = 1 - \frac{10}{\lambda^c}.\end{aligned}$$

The reason that $\Pr[\widetilde{K} = K \mid \mathsf{Evnt}_1 \wedge \mathsf{Agree} \wedge \overline{\mathsf{Empty}}] = 1$ is that, the oracle $\widetilde{\mathbf{O}} := (\widetilde{\mathbf{g}}, \mathbf{e}, \widetilde{\mathbf{d}})$, defined in Line 4b of Brk, is PKE-valid (c.f., Lemma 4.11). Also, (C, K) is a possible output of $\mathrm{Comm}^{\widetilde{\mathbf{O}}}(\mathsf{PK}_1, \ldots, \mathsf{PK}_n)$, since Comm makes only encryption queries. Let $h \in [n]$ be the index for which Evnt_1 holds. Since Agree occurs, CRS and $(\mathsf{PK}_i, *)$ for $i \neq h$ are a possible output of $\mathrm{CRSGen}^{\widetilde{\mathbf{g}}}(1^\lambda)$ and $\mathrm{Init}^{\widetilde{\mathbf{g}}}(\mathsf{CRS})$, respectively. Also, we know $(\widetilde{\mathsf{SK}}_h, \mathsf{PK}_h)$ is a possible output of $\mathrm{Init}^{\widetilde{\mathbf{g}}}(\mathsf{CRS})$, because $\widetilde{\mathbf{g}}$ and $\mathbf{g}'_h$ agree with each other. Now since $\mathsf{Evnt}_1 \wedge \overline{\mathsf{Empty}}$ holds, this means that $\overline{\mathsf{Evnt}_2} \wedge \overline{\mathsf{Empty}}$ holds, which means Line 4(c)iii of Brk is never hit, and so the simulated decryption performed by Brk (for the index h) results in the same value as $\mathrm{Derive}^{\widetilde{\mathbf{d}}}(\widetilde{\mathsf{SK}}_h, \mathsf{PK}_1, \ldots, \mathsf{PK}_n, C)$. The proof is now complete. □

5 No *Single* Optimal CGKA Protocol Exists

In this section, we will show that there is no *single best* CGKA protocol. More precisely, for any CGKA protocol Π, there is a distribution of CGKA sequences and some other CGKA protocol

$\mathsf{\Pi}'$ such that on sequences drawn from this distribution, $\mathsf{\Pi}'$ has much lower expected amortized communication cost than $\mathsf{\Pi}$. We make the same restriction on protocols that we have throughout the paper: the protocols are only allowed to use PKE.

The main intuition behind this section is the following: As we saw from Corollary 3.2 of Theorem 3.1, if starting with a group of n users with public keys $\mathsf{PK}_1, \ldots \mathsf{PK}_n$ in any state (for example, every user has just executed an update),

1. k users are added to the group and then remain offline (i.e., do not execute any operations),

2. Then the α users (w.l.o.g., users $1, \ldots, \alpha$ with public keys $\mathsf{PK}_1, \ldots \mathsf{PK}_\alpha$) that have been online since the first of the above users was added all update,

the combined size of their ciphertexts must be $\Omega(k)$. Now, consider the scenario in which user 1 adds all of the k new users, then updates, and then users $2, \ldots, \alpha$ all execute updates. While adding the k new users, user 1 may or may not have built some structure for group members to communicate with them until they come online (for example, in TTKEM, c.f. Appendix F, user 1 would have sampled and communicated key pairs for all nodes that are on the paths from the k users' leaves to the root). The protocol $\mathsf{\Pi}$ is then left with a choice regarding the updates of users $2, \ldots, \alpha$. Roughly, either:

(a) Each of the users $2, \ldots, \alpha$ rebuild complete structure themselves (say, sample and communicate their own key pairs for nodes on the paths from the k users' leaves to the root, as user 1 would have done when adding them in TTKEM) to communicate with the k newly added users; or

(b) At least one such user i does not (i.e., they only rebuild asymptotically incomplete structure themselves) and thus relies on some asymptotically non-trivial amount of structure created by the users that have executed operations before them to communicate with the k added users.

We will however show that both (a) and (b) can be losing strategies; i.e., no matter if a protocol $\mathsf{\Pi}$ chooses strategy (a) or (b) (or probabilistically favors one over the other), it can be starkly outperformed by another protocol $\mathsf{\Pi}'$ when executing certain sequences (by the same amount in both cases). In the case of (a), if after users $2, \ldots, \alpha$ execute their updates, the k added users come online and execute their own updates, then users $2, \ldots, \alpha$ all rebuilt complete structure themselves unnecessarily – the k added users can themselves create structure which allows others to communicate with them thereafter using $O(\log n)$ communication each (for example, in TTKEM, they would just sample key pairs for their paths). Therefore if all subsequent operations are updates, the communication of the protocol can easily stay low. So, if $\mathsf{\Pi}$ chose (a) then it communicated a factor of $\Omega(k/\log n)$ more than it had to during the updates of Step 2; or $\Omega(n/\log n)$ if $k = \Omega(n)$. In Section 5.1, we formally define the distribution containing such sequences as $\mathsf{ActiveBad}$ and in Section 5.2 formally prove the statement of the previous sentence. (Technically, for fairness reasons when comparing with the result of the next paragraph, we also account for the communication of a certain number of updates after Step 2. So the result, while qualitatively the same, is quantitatively not as stark.)

In the case of (b) consider the scenario in which (i) one of the α active users, user j, is randomly selected to become *passive* for the remainder of the sequence, i.e., they never execute another operation, then (ii) the other $\alpha - 1$ active users perform ℓ rounds of taking turns executing updates.

If Π chose strategy (b) and user j is the one who only rebuilt asymptotically incomplete structure themselves, then according to Corollary 3.2, each of the ℓ rounds of Step (ii) will have high $\Omega(k)$ communication each. However, if strategy (a) had been chosen by Π (and user 1 built complete structure as well) then the communication of user j would allow for the ℓ rounds of Step (ii) to be executed with low communication: $O(\alpha \log n)$ (using TTKEM-like updates; we explain more later). So if Π chose (b) then in expectation, it communicated a factor of $\Omega(\ell k/(\alpha \cdot (k\alpha + \ell\alpha \log n)))$ more than it had to; or $\Omega(n/\log n)$ if $k = \Omega(n)$, $\ell = \Theta(n/\log n)$, and $\alpha = O(1)$. In Section 5.1, we formally define the distribution containing such sequences as $\mathsf{LazyBad}$ and in Section 5.2 formally prove the statement of the previous sentence (albeit with slightly different concrete parameters for k, ℓ, and α).

5.1 Bad Sequences of Operations

We first formally define the two distributions of sequences, $\mathsf{LazyBad}$ and $\mathsf{ActiveBad}$, such that for any CGKA protocol Π, we can choose one of these distributions and it will be the case that there is some Π' which has much lower expected communication than Π on that distribution. Both $\mathsf{LazyBad}$ and $\mathsf{ActiveBad}$ are parameterized by:

- n: The number of users in the group before user 1 adds the new users;
- $\mathsf{PreAddSeq}$: The operations of the pre-add phase, i.e., the sequence of *valid* operations (the first operation is Add to create the group, only users that are not in the group are added by users in the group, only users in the group are removed by other users in the group, only users in the group can execute an update, and at the end of Seq the group has n members) to be executed before the k adds and subsequent operations of $\mathsf{ActiveBad}$ or $\mathsf{LazyBad}$.
- k: The number of users added by user 1;
- α: the number of active users after the first of the k users is added; and
- ℓ: For $\mathsf{LazyBad}$, the number of rounds of updates in which one of the originally active users is passive. We use ℓ in $\mathsf{ActiveBad}$ only to ensure that on input the same parameters, the two types of sequences have the same length (for fairness reasons).

We define both types of sequences as distributions, even though $\mathsf{ActiveBad}(n, \mathsf{PreAddSeq}, k, \alpha, \ell)$ is just one sequence (i.e., that sequence is drawn from the distribution $\mathsf{ActiveBad}(n, \mathsf{PreAddSeq}, k, \alpha, \ell)$ with probability 1). In the following, we will assume that both n and k are powers of 2, for simplicity. Also, we will often make the parameters n, k, α, and ℓ implicit and simply refer to $\mathsf{ActiveBad}(n, \mathsf{PreAddSeq}, k, \alpha, \ell)$ as $\mathsf{ActiveBad}(\mathsf{PreAddSeq})$ and $\mathsf{LazyBad}(n, \mathsf{PreAddSeq}, k, \alpha, \ell)$ as $\mathsf{LazyBad}(\mathsf{PreAddSeq})$. We first define $\mathsf{LazyBad}(\mathsf{PreAddSeq})$:

Definition 5.1. *A sequence* Seq *of CGKA operations drawn from distribution* $\mathsf{LazyBad}(n, \mathsf{PreAddSeq}, k, \alpha, \ell)$ *consists of the following phases:*

- **Phase** $\mathsf{P0}$*: The pre-add phase, i.e., the operations* $\mathsf{Op}_1, \ldots, \mathsf{Op}_{t_1^A - 1}$ *of* $\mathsf{PreAddSeq}$.
- **Phase** $\mathsf{P1}$*: For* $i \in [k]$ *operations* $\mathsf{Op}_{1,i} = (\mathrm{Add}, \mathsf{PK}_1, \mathsf{PK}_{n+i})$*. Then operation* $\mathsf{Op}_{1,k+1} = (\mathrm{Up}, \mathsf{PK}_1, \perp)$.
- **Phase** $\mathsf{P2}$*: For* $i \in [\alpha - 1]$ *operations* $\mathsf{Op}_{2,i} = (\mathrm{Up}, \mathsf{PK}_{i+1}, \perp)$.

- **Phase** $\mathsf{P3}$*: Let* $j \leftarrow_{\$} [\alpha]$*. Then, for each* $m \in [\ell]$*: for every* $i < j$ *(resp.* $i > j$*),* $\mathsf{Op}_{3,(m-1)(\alpha-1)+i} = (\mathrm{Up}, \mathsf{PK}_i, \bot)$ *(resp.* $\mathsf{Op}_{3,(m-1)(\alpha-1)+i-1} = (\mathrm{Up}, \mathsf{PK}_i, \bot)$*), where* PK_i *is the most recent public key of user* i*.*

Next, we define $\mathsf{ActiveBad}(\mathsf{PreAddSeq})$, which has the same phases $0-2$ as $\mathsf{LazyBad}(\mathsf{PreAddSeq})$, but differs in phase 3 as described above:

Definition 5.2. *A sequence* Seq *of CGKA operations drawn from distribution* $\mathsf{ActiveBad}(n, \mathsf{PreAddSeq}, k, \alpha, \ell)$ *consists of the same phases* $\mathsf{P0}$*-*$\mathsf{P2}$ *as above then:*

- **Phase** $\mathsf{P3}$*: For* $i \in [\ell \cdot (\alpha-1)]$*:* $\mathsf{Op}_{3,i} = (\mathrm{Up}, \mathsf{PK}_{n+1+(i \bmod \alpha)}, \bot)$*, where* $\mathsf{PK}_{n+1+(i \bmod \alpha)}$ *is the most recent public key of user* $n+1+(i \bmod \alpha)$*.*

Note that by Theorem 3.1, for every CGKA protocol it must be that update $\mathsf{Op}_{1,k+1} = (\mathrm{Up}, \mathsf{PK}_1, \bot)$ in Phase $\mathsf{P1}$ of either distribution requires $\Omega(k)$ communication, no matter what the operations of $\mathsf{PreAddSeq}$ were and what structure the adds of user 1 in Phase $\mathsf{P1}$ created. Since with $O(k)$ communication, user 1 can in this update create full structure with which other users in the group can communicate with the added $\mathsf{PK}_{n+1} \ldots, \mathsf{PK}_{n+\alpha}$ thereafter (as in TTKEM), it is intuitively the best choice for a protocol to use this behavior for user 1. Thus, since we aim to define these two distributions in a way that emphasizes the different choices protocols can make to minimize communication, user 1's first update is included in Phase $\mathsf{P1}$ and we define the communication complexity of a protocol executing a sequence drawn from one of these two distributions to include only the communication costs of the operations in Phase $\mathsf{P2}$ and $\mathsf{P3}$:

Definition 5.3. *Let* Seq *be a sequence of CGKA operations drawn from distribution* $\mathsf{LazyBad}(\mathsf{PreAddSeq})$ *(resp.* $\mathsf{ActiveBad}(\mathsf{PreAddSeq})$*) and* $\mathsf{CC}_\Pi[\mathsf{Op}]$ *be the communication cost of a CGKA protocol* Π *executing operation* Op *of* Seq *after executing all preceding operations of* Seq *in order. Then:*

1. *The* amortized communication complexity *of a protocol* Π *that executes* Seq *is* $\mathbf{CC}_\Pi[\mathsf{Seq}] := (\sum_{\mathsf{Op} \in \mathsf{P2} \cup \mathsf{P3}} \mathsf{CC}_\Pi[\mathsf{Op}])/((\alpha-1)\cdot(\ell+1))$*, where* $\mathsf{P2}$ *and* $\mathsf{P3}$ *are the corresponding phases in* Seq *of* $\mathsf{LazyBad}(\mathsf{PreAddSeq})$ *(resp.* $\mathsf{ActiveBad}(\mathsf{PreAddSeq})$*).*

2. *The* expected amortized communication complexity *of a protocol* Π *on random* Seq *drawn from* $\mathsf{LazyBad}(\mathsf{PreAddSeq})$ *(resp.* $\mathsf{ActiveBad}(\mathsf{PreAddSeq})$*) is*

$$\mathbf{CC}_\Pi(\mathsf{LazyBad}(\mathsf{PreAddSeq})) := \mathbb{E}_{\mathsf{Seq} \leftarrow_{\$} \mathsf{LazyBad}(\mathsf{PreAddSeq})}[\mathbf{CC}_\Pi[\mathsf{Seq}]]$$

$$(\textit{resp. } \mathbf{CC}_\Pi(\mathsf{ActiveBad}(\mathsf{PreAddSeq})) := \mathbb{E}_{\mathsf{Seq} \leftarrow_{\$} \mathsf{ActiveBad}(\mathsf{PreAddSeq})}[\mathbf{CC}_\Pi[\mathsf{Seq}]]),$$

where the randomness is over the choice of Seq *and the random coins of* Π*.*

5.2 Suboptimality of all CGKA Protocols

We now state and prove our Theorem showing that all CGKA protocols must have suboptimal expected amortized communication complexity on either $\mathsf{LazyBad}(\mathsf{PreAddSeq})$ or $\mathsf{ActiveBad}(\mathsf{PreAddSeq})$. First, we define a specific $\mathsf{PreAddSeq}$ which intuitively leaves the CGKA group in a *full* state:

Definition 5.4. *Valid sequence of CGKA operations* Full_n *contains the following operations in order:* $(\mathrm{Add}, \mathsf{PK}_1, \mathsf{PK}_2), (\mathrm{Add}, \mathsf{PK}_1, \mathsf{PK}_3), \ldots, (\mathrm{Add}, \mathsf{PK}_1, \mathsf{PK}_n), (\mathrm{Up}, \mathsf{PK}_1, \bot), (\mathrm{Up}, \mathsf{PK}_2, \bot), \ldots, (\mathrm{Up}, \mathsf{PK}_n, \bot)$*.*

Theorem 5.5. *Let $\ell = O(k/\log n)$. Then for every CGKA protocol $\mathsf{\Pi}$ and every* $\mathsf{PreAddSeq}$*, there exists some other protocol $\mathsf{\Pi}'$ such that either*

$$\mathbf{CC}_{\mathsf{\Pi}}(\mathsf{LazyBad}(\mathsf{PreAddSeq})) \geq \mathbf{CC}_{\mathsf{\Pi}'}(\mathsf{LazyBad}(\mathsf{Full}_n)) \cdot \Omega(\ell/\alpha^2), \text{ or}$$

$$\mathbf{CC}_{\mathsf{\Pi}}(\mathsf{ActiveBad}(\mathsf{PreAddSeq})) \geq \mathbf{CC}_{\mathsf{\Pi}'}(\mathsf{ActiveBad}(\mathsf{Full}_n)) \cdot \Omega(k/\ell \log n).$$

Note that $\mathsf{PreAddSeq}$ can be any *valid* sequence that results in a group with n members, including (but not limited to) Full_n. As will be seen, our results combine general lower bounds for the considered protocol $\mathsf{\Pi}$ on any $\mathsf{PreAddSeq}$, with upper bounds for protocols $\mathsf{\Pi}'$ on specifically Full_n.

Before proving the Theorem, we separate CGKA protocols $\mathsf{\Pi}$ into two classes based on their expected behavior in phase $\mathsf{P2}$ of a sequence drawn from $\mathsf{LazyBad}(\mathsf{PreAddSeq})$ or $\mathsf{ActiveBad}(\mathsf{PreAddSeq})$. The first class of protocols are more likely than not to have some *lazy* user in phase $\mathsf{P2}$: i.e., a user whose update operation $\mathsf{Op}_{2,i} = (\mathrm{Up}, \mathsf{PK}_{i+1}, \perp)$ in phase $\mathsf{P2}$ has communication cost $\mathsf{CC}_{\mathsf{\Pi}}[\mathsf{Op}] = o(k)$. The other class of protocols are the opposite – they are more likely than not to have only *heavy* users in phase $\mathsf{P2}$: i.e., all users have update operations $\mathsf{Op}_{2,i} = (\mathrm{Up}, \mathsf{PK}_{i+1}, \perp)$ in phase $\mathsf{P2}$ with communication cost $\mathsf{CC}_{\mathsf{\Pi}}[\mathsf{Op}] = \Omega(k)$.

Definition 5.6. *CGKA protocol $\mathsf{\Pi}$ is* Lazy *if* $\Pr[\exists i \in [\alpha - 1] : \mathbf{CC}_{\mathsf{\Pi}}[\mathsf{Op}_{2,i}] = o(k)] > 1/2$*. Otherwise, $\mathsf{\Pi}$ is* Active*.*

We first show that there is a protocol $\mathsf{\Pi}_{\mathsf{Active}}$ that has efficient communication on sequences drawn from $\mathsf{LazyBad}(\mathsf{Full}_n)$.

Lemma 5.7. *There is a protocol $\mathsf{\Pi}_{\mathsf{Active}}$ that has expected amortized communication cost $\mathbf{CC}_{\mathsf{\Pi}_{\mathsf{Active}}}(\mathsf{LazyBad}(\mathsf{Full}_n)) = O(k/\ell + \log n)$ on random* Seq *drawn from* $\mathsf{LazyBad}(n, \mathsf{Full}_n, k, \alpha, \ell)$*.*

Proof. The protocol $\mathsf{\Pi}_{\mathsf{Active}}$ simply executes in phases $\mathsf{P0}$ and $\mathsf{P1}$ as TTKEM does (c.f. Appendix F). It is easy to see that for any Seq drawn from $\mathsf{LazyBad}(n, \mathsf{Full}_n, k, \alpha, \ell)$, after phase $\mathsf{P1}$ all nodes on the paths of added users' $(\mathsf{PK}_{n+1}, \ldots, \mathsf{PK}_{n+k})$ leaves to the root are tainted by user 1, and all other nodes are untainted. Then, in phase $\mathsf{P2}$ each of the users that execute $\mathrm{Up}(\mathsf{PK}_i)$ behave as user 1 did in phase $\mathsf{P1}$, except that they refresh those nodes that are on the direct path of their own leaf, instead of user 1's leaf: i.e., they each independently refresh the tainted nodes of user 1 (those on the paths from the leaves corresponding to $\mathsf{PK}_{n+1}, \ldots, \mathsf{PK}_{n+k}$) in addition to the nodes on their direct path. Since there are $O(k + \log n)$ such nodes, it can easily be seen that each such $\mathrm{Up}(\mathsf{PK}_i)$ can be done with communication cost $\mathsf{CC}_{\mathsf{\Pi}}[\mathsf{Op}] = O(k + \log n)$ (by systematically generating new secrets for each node and decrypting it to the public keys of its children, from the bottom of the tree to the top) and thus the total communication cost of phase $\mathsf{P2}$ is $O((k + \log n) \cdot \alpha)$. Then, in phase $\mathsf{P3}$, the users that execute $\mathrm{Up}(\mathsf{PK}_i)$ in each of the ℓ repetitions simply refresh the nodes on their direct path and use the public keys generated by the user j that remains passive in phase $\mathsf{P3}$ (i.e., does not execute any operations) to communicate with the added users $\mathsf{PK}_{n+1}, \ldots, \mathsf{PK}_{n+k}$; all other key pairs on the added users' paths are never again used. Therefore, the total communication cost of phase $\mathsf{P3}$ is $O(\ell \cdot \alpha \cdot \log n)$. Thus, $\mathbf{CC}_{\mathsf{\Pi}_{\mathsf{Active}}}(\mathsf{LazyBad}) = O(k/\ell + \log n)$.

The security of $\mathsf{\Pi}_{\mathsf{Active}}$ follows almost immediately from the security of TTKEM and thus we omit a formal proof for brevity. Informally, the security of phases $\mathsf{P0}$ and $\mathsf{P1}$ follows directly from the security of TTKEM. Now, assume that when phase $\mathsf{P2}$ begins, all users outside of users $1, \ldots, \alpha$ that have been corrupted by the adversary have since executed an Up operation and they are never corrupted by the adversary again. If this is not the case, then security of the subsequent operations is not required for any CGKA protocol Π since, by correctness, the adversary can recover all group secrets of these operations. There are two scenarios to consider: First, if the chosen passive user j is corrupted after its update $\mathrm{Up}(\mathsf{PK}_j)$ of phase $\mathsf{P2}$, then anyway for any CGKA protocol Π, security of the operations in phase $\mathsf{P3}$ is not required, as above. Otherwise, if j is not corrupted after its update $\mathrm{Up}(\mathsf{PK}_j)$ of phase $\mathsf{P2}$, then the key pairs that it generates for those nodes that are on the paths from the leaves corresponding to $\mathsf{PK}_{n+1}, \ldots, \mathsf{PK}_{n+k}$ remain secure. Additionally, the users that execute operations $\mathrm{Up}(\mathsf{PK}_i)$ in phase $\mathsf{P3}$ simply no longer encrypt to the key pairs generated by users m other than user j for these nodes, so even if some such user m is corrupted, no secrets are encrypted to such key pairs it has generated. Moreover, they otherwise execute their update according to TTKEM, so their updates facilitate recovery as in TTKEM, and thus security follows. □

Now we show that those protocols Π that are Lazy do not have efficient communication on sequences drawn from $\mathsf{LazyBad}(\mathsf{PreAddSeq})$ for any $\mathsf{PreAddSeq}$.

Lemma 5.8. *For every protocol Π that is Lazy and every $\mathsf{PreAddSeq}$, the expected total communication cost $\mathbf{CC}_{\Pi}(\mathsf{LazyBad}(\mathsf{PreAddSeq})) = \Omega(k/\alpha^2)$ on random Seq drawn from $\mathsf{LazyBad}(n, \mathsf{PreAddSeq}, k, \alpha, \ell)$.*

Proof. Since Π is Lazy, with probability greater than $1/2$, one of the users i who executes $\mathrm{Up}(\mathsf{PK}_i)$ in phase $\mathsf{P2}$ does so with communication cost $\mathsf{CC}_{\Pi}[\mathsf{Op}] = o(k)$. The probability that this user is the user j randomly chosen in phase $\mathsf{P3}$ to remain passive (i.e., not execute any more operations) for the rest of Seq is $1/\alpha$. If this is indeed the case, then by Corollary 3.2, since the update of PK_i had communication cost $o(k)$, we know that each of the ℓ repetitions of phase $\mathsf{P3}$ will have total communication cost $\Omega(k)$. Putting things together, we have that for Lazy protocols Π, $\mathbf{CC}_{\Pi}(\mathsf{LazyBad}(\mathsf{PreAddSeq})) > \frac{1}{2\alpha} \cdot (\ell \cdot \Omega(k))/O(\alpha \cdot \ell) = \Omega(k/\alpha^2)$. □

Next we show that there is a protocol Π_{Lazy} that has efficient communication on sequences drawn from $\mathsf{ActiveBad}(\mathsf{Full}_n)$.

Lemma 5.9. *There is a protocol Π_{Lazy} that has expected total communication cost $\mathbf{CC}_{\Pi_{\mathsf{Lazy}}}(\mathsf{ActiveBad}(\mathsf{Full}_n)) = O(\log n)$ on random Seq drawn from $\mathsf{ActiveBad}(n, \mathsf{Full}_n, k, \alpha, \ell)$.*

Proof. Π_{Lazy} is simply TTKEM (c.f. Appendix F): It is easy to see that for any Seq drawn from $\mathsf{ActiveBad}(n, \mathsf{Full}_n, k, \alpha, \ell)$, after phase $\mathsf{P1}$ all nodes that are on the paths of added users' $(\mathsf{PK}_{n+1}, \ldots, \mathsf{PK}_{n+k})$ leaves to the root are tainted by user 1, and all other nodes are untainted. Therefore, it is obvious that all operations $\mathrm{Up}(\mathsf{PK}_i)$ of $\mathsf{P2}$ and $\mathsf{P3}$ have communication cost $\mathsf{CC}_{\Pi}[\mathsf{Op}] = O(\log n)$, since all such executing users own 0 taints. Thus $\mathbf{CC}_{\Pi_{\mathsf{Lazy}}}(\mathsf{ActiveBad}) = O(\log n)$. □

Finally, we show that those protocols Π that are Active do not have efficient communication on sequences drawn from $\mathsf{ActiveBad}(\mathsf{PreAddSeq})$ for any $\mathsf{PreAddSeq}$.

Lemma 5.10. *For every protocol Π that is Active and every $\mathsf{PreAddSeq}$, its expected total communication cost $\mathbf{CC}_{\Pi}(\mathsf{ActiveBad}(\mathsf{PreAddSeq})) = \Omega(k/\ell)$ on random Seq drawn from $\mathsf{ActiveBad}(n, \mathsf{PreAddSeq}, k, \alpha, \ell)$.*

Proof. Since $\mathsf{\Pi}$ is Active, with probability at least $1/2$, all of the users i that execute operations $\mathrm{Up}(\mathsf{PK}_i)$ in phase $\mathsf{P2}$ do so with communication cost $\mathsf{CC}_{\mathsf{\Pi}}[\mathsf{Op}] = \Omega(k)$. If this is the case, then we know that phase $\mathsf{P2}$ has total communication cost $\Omega(\alpha \cdot k)$. Putting things together, we have that for Active protocols $\mathsf{\Pi}$, $\mathbf{CC}_{\mathsf{\Pi}}(\mathsf{ActiveBad}(\mathsf{PreAddSeq})) \geq \frac{1}{2} \cdot \Omega(\alpha \cdot k)/O(\alpha \cdot \ell) = \Omega(k/\ell)$. □

Proof of Theorem 5.5. Combining the results of Lemmas 5.7, 5.8, 5.9, and 5.10, Theorem 5.5 easily follows. □

The following corollary thus easily follows:

Corollary 5.11. *Let $k = \Omega(n)$, $\ell = \Theta(\sqrt{n})$, and $\alpha = O(\sqrt{\log n})$. Then for every protocol $\mathsf{\Pi}$, there exists some other protocol $\mathsf{\Pi}'$ such that either on a random sequence drawn from $\mathsf{ActiveBad}(\mathsf{Full}_n)$, or from $\mathsf{LazyBad}(\mathsf{Full}_n)$, $\mathsf{\Pi}'$ has a factor of $\Omega(\sqrt{n}/\log n)$ better amortized communication in expectation than $\mathsf{\Pi}$ does.*

References

[1] Alwen, J., Auerbach, B., Noval, M.C., Klein, K., Pascual-Perez, G., Pietrzak, K., Walter, M.: Cocoa: Concurrent continuous group key agreement. In: Advances in Cryptology - EUROCRYPT 2022 (2022)

[2] Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part I. LNCS, vol. 11476, pp. 129–158. Springer, Heidelberg (May 2019)

[3] Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the IETF MLS standard for group messaging. In: Micciancio, D., Ristenpart, T. (eds.) CRYPTO 2020, Part I. LNCS, vol. 12170, pp. 248–277. Springer, Heidelberg (Aug 2020)

[4] Alwen, J., Coretti, S., Jost, D., Mularczyk, M.: Continuous group key agreement with active security. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 261–290. Springer, Heidelberg (Nov 2020)

[5] Alwen, J., Jost, D., Mularczyk, M.: On the insider security of mls. Cryptology ePrint Archive, Report 2020/1327 (2020), https://eprint.iacr.org/2020/1327

[6] Alwen, J., Capretto, M., Cueto, M., Kamath, C., Klein, K., Markov, I., Pascual-Perez, G., Pietrzak, K., Walter, M., Yeo, M.: Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement. In: 2021 IEEE Symposium on Security and Privacy (SP). IEEE (2021)

[7] Balli, F., Rösler, P., Vaudenay, S.: Determining the core primitive for optimally secure ratcheting. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020, Part III. LNCS, vol. 12493, pp. 621–650. Springer, Heidelberg (Dec 2020)

[8] Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., Cohn-Gordon, K.: The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-14, Internet Engineering Task Force (May 2022), https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-14, work in Progress

[9] Bhargavan, K., Barnes, R., Rescorla, E.: TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups (2018), pubs/treekem.pdf, published at https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8

[10] Bienstock, A., Dodis, Y., Garg, S., Grogan, G., Hajiabadi, M., Rösler, P.: On the worst-case inefficiency of CGKA. In: TCC 2022. LNCS, Springer (2022)

[11] Bienstock, A., Dodis, Y., Rösler, P.: On the price of concurrency in group ratcheting protocols. In: Pass, R., Pietrzak, K. (eds.) TCC 2020, Part II. LNCS, vol. 12551, pp. 198–228. Springer, Heidelberg (Nov 2020)

[12] Bienstock, A., Dodis, Y., Tang, Y.: Multicast key agreement, revisited. In: Galbraith, S.D. (ed.) Topics in Cryptology – CT-RSA 2022. pp. 1–25. Springer International Publishing, Cham (2022)

[13] Bienstock, A., Dodis, Y., Yeo, K.: Forward secret encrypted ram: Lower bounds and applications. In: TCC 2021: 19th Theory of Cryptography Conference (2021)

[14] Bienstock, A., Fairoze, J., Garg, S., Mukherjee, P., Raghuraman, S.: A more complete analysis of the signal double ratchet algorithm. Cryptology ePrint Archive, Report 2022/355 (2022), https://ia.cr/2022/355

[15] Boneh, D., Papakonstantinou, P.A., Rackoff, C., Vahlis, Y., Waters, B.: On the impossibility of basing identity based encryption on trapdoor permutations. In: 49th FOCS. pp. 283–292. IEEE Computer Society Press (Oct 2008)

[16] Boneh, D., Zhandry, M.: Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014, Part I. LNCS, vol. 8616, pp. 480–499. Springer, Heidelberg (Aug 2014)

[17] Canetti, R., Garay, J., Itkis, G., Micciancio, D., Naor, M., Pinkas, B.: Multicast security: a taxonomy and some efficient constructions. In: IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No.99CH36320). vol. 2, pp. 708–716 vol.2 (1999)

[18] Canetti, R., Jain, P., Swanberg, M., Varia, M.: Universally composable end-to-end secure messaging. Cryptology ePrint Archive, Report 2022/376 (2022), https://ia.cr/2022/376

[19] Canetti, R., Kalai, Y.T., Paneth, O.: On obfuscation with random oracles. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015, Part II. LNCS, vol. 9015, pp. 456–467. Springer, Heidelberg (Mar 2015)

[20] Chung, K.M., Lin, H., Mahmoody, M., Pass, R.: On the power of nonuniformity in proofs of security. In: Kleinberg, R.D. (ed.) ITCS 2013. pp. 389–400. ACM (Jan 2013)

[21] Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. In: 2017 IEEE European Symposium on Security and Privacy (EuroS P). pp. 451–466 (2017)

[22] Cohn-Gordon, K., Cremers, C., Garratt, L., Millican, J., Milner, K.: On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018. pp. 1802–1819. ACM Press (Oct 2018)

[23] Coretti, S., Dodis, Y., Guo, S., Steinberger, J.P.: Random oracles and non-uniformity. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part I. LNCS, vol. 10820, pp. 227–258. Springer, Heidelberg (Apr / May 2018)

[24] Dodis, Y., Guo, S., Katz, J.: Fixing cracks in the concrete: Random oracles with auxiliary input, revisited. In: Coron, J.S., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part II. LNCS, vol. 10211, pp. 473–495. Springer, Heidelberg (Apr / May 2017)

[25] Dowling, B., Hauck, E., Riepel, D., Rösler, P.: Strongly anonymous ratcheted key exchange. In: ASIACRYPT 2022. LNCS (2022)

[26] Garg, S., Hajiabadi, M., Mahmoody, M., Mohammed, A.: Limits on the power of garbling techniques for public-key encryption. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part III. LNCS, vol. 10993, pp. 335–364. Springer, Heidelberg (Aug 2018)

[27] Gennaro, R., Trevisan, L.: Lower bounds on the efficiency of generic cryptographic constructions. In: 41st FOCS. pp. 305–313. IEEE Computer Society Press (Nov 2000)

[28] Harney, H., Muckenhirn, C.: Rfc2093: Group key management protocol (gkmp) specification (1997)

[29] Mittra, S.: Iolus: A framework for scalable secure multicasting. In: Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. p. 277–288. SIGCOMM '97, Association for Computing Machinery, New York, NY, USA (1997), https://doi.org/10.1145/263105.263179

[30] Perrin, T., Marlinspike, M.: The double ratchet algorithm (2016), https://signal.org/docs/specifications/doubleratchet/

[31] Poettering, B., Rösler, P., Schwenk, J., Stebila, D.: SoK: Game-based security models for group key exchange. In: Paterson, K.G. (ed.) CT-RSA 2021. LNCS, vol. 12704, pp. 148–176. Springer, Heidelberg (May 2021)

[32] Rösler, P., Mainka, C., Schwenk, J.: More is less: On the end-to-end security of group chats in signal, whatsapp, and threema. In: 2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018 (2018)

[33] Sherman, A.T., McGrew, D.A.: Key establishment in large dynamic groups using one-way function trees. IEEE Transactions on Software Engineering 29(5), 444–458 (2003)

[34] Smart, N.P.: Efficient key encapsulation to multiple parties. In: Blundo, C., Cimato, S. (eds.) SCN 04. LNCS, vol. 3352, pp. 208–219. Springer, Heidelberg (Sep 2005)

[35] Wallner, D., Harder, E., Agee, R.: Rfc2627: Key management for multicast: Issues and architectures (1999)

[36] Weidner, M., Kleppmann, M., Hugenroth, D., Beresford, A.R.: Key agreement for decentralized secure group messaging with strong security guarantees. In: Vigna, G., Shi, E. (eds.) ACM CCS 2021. pp. 2024–2045. ACM Press (Nov 2021)

[37] Wong, C.K., Gouda, M., Lam, S.S.: Secure group communications using key graphs. In: Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. p. 68–79. SIGCOMM '98, Association for Computing Machinery, New York, NY, USA (1998), https://doi.org/10.1145/285237.285260

A Omitted Preliminaries

Lemma A.1. *Let $x_1, \ldots, x_r$ be independent boolean random variables where for all $i \in [r]$, $\Pr[x_i = 1] \geq p$. Then, $\Pr[x_1 = \cdots = x_r = 0] \leq \frac{1}{e^{pr}}$.*

Proof. For all real x: $1 + x \leq e^x$, and hence $(1 - p) \leq e^{-p}$. □

The black-box attack against CKE protocols makes use of the following lemma about output compression with respect to random oracles. The proof of this result uses a simple adaptation of the output compression technique from [24] (which, in turn, goes back to [27]), developed in the context of the auxiliary random oracle model. We simply tweak the parameters to match our precise setting.

Lemma A.2. *Let $\mathsf{A} = (\mathsf{A}_0^{\mathbf{g}}, \mathsf{A}_1^{\mathbf{e}})$ be a two-phase adversary, where $\mathsf{A}_0^{\mathbf{g}}(1^\lambda)$ outputs a string x_0 while only calling $\mathbf{g}$, and $\mathsf{A}_1^{\mathbf{e}}(x_0)$ outputs a string x_1 while only calling $\mathbf{e}$. Let $\mathsf{B}^{\mathbf{g},\mathbf{u},\mathbf{v},\mathbf{d}}(x_0, x_1)$ be an adversary that takes as input (x_0, x_1), makes some number of queries to $(\mathbf{g}, \mathbf{u}, \mathbf{v}, \mathbf{d})$ (but not to $\mathbf{e}$) and outputs a set $\mathsf{Chal} = \{(\mathsf{pk}_1, c_1), \ldots, (\mathsf{pk}_w, c_w)\}$. We say the event $\mathsf{Success}$ holds if (i) $w \geq \lceil 2^{\frac{|x_1|}{3\lambda}} \rceil + 1$; (ii) all the pairs are distinct, and (iii) for all $i \in [w]$ $\mathbf{v}(\mathsf{pk}_i, c_i) = \top$. We then have $\Pr[\mathsf{Success}] \leq 2^{-\lambda/2} = \mathsf{negl}(\lambda)$, where the probability is taken over $(\mathbf{g}, \mathbf{e}, \mathbf{d}, \mathbf{u}, \mathbf{v}) \leftarrow_\$ \Psi$ and the random coins of A and B.*

Proof. (Sketch) As in all output compression results, we will show that the existence of such algorithms A, B, where B's probability of success is $\varepsilon = 2^{-\lambda/2}$, leads to a pair of (computationally unbounded) compression/decompression algorithms C and D which successfully compress a random function $\mathbf{e} : \{0,1\}^{3\lambda} \times \{0,1\} \times \{0,1\}^\lambda \to \{0,1\}^{3\lambda}$ in a provably impossible way. Namely: (1) the output of $C(\mathbf{e})$ will save more than $\log(1/\varepsilon)$ bits compared to the length of $\mathbf{e}$; (3) D can successfully recover $\mathbf{e}$ with probability $1 - \varepsilon$ on the ("short") output of the compressor $C(\mathbf{e})$.

In our case, we can fix random oracles $(\mathbf{g}, \mathbf{u}, \mathbf{v}, \mathbf{d})$ and local randomness of A, B, and only focus on compressing $\mathbf{e}$. Our compressor $C(\mathbf{e})$ will emulate the run A, B, and will only succeed when A, B jointly produce w values $\mathsf{Chal} = \{(\mathsf{pk}_1, c_1), \ldots, (\mathsf{pk}_w, c_w)\}$ satisfying conditions (i)-(iii) of the Lemma. In particular, condition (iii) implies that for each of the w values (pk_i, c_i) there exists inputs $(m_i, r_i) \in \{0,1\} \times \{0,1\}^\lambda$ s.t. $\mathbf{e}(\mathsf{pk}_i, m_i, r_i) = c_i$. Our compressor $C(\mathbf{e})$ will then output: (a) the compression string x_1 output by $\mathsf{A}_1^{\mathbf{e}}(x_0)$; (b) the set $\mathsf{In} = \{(m_1, r_1), \ldots, (m_w, r_w)\}$ in lexicographic order; (c) in lexicographic order, the set of all other outputs of $\mathbf{e}$ except for w "special" input points $(\mathsf{pk}_i, m_i, r_i)$.

We need to check two things now. First, that the output of $C(\mathbf{e})$ is enough for D to reconstruct $\mathbf{e}$ (when A, B succeed). Second, that the output of $C(\mathbf{e})$ is too short under the conditions of the Lemma. For the first part, notice that D has all the information it needs to successfully complete the run of A, B, and thus obtain the set $\mathsf{Chal} = \{(\mathsf{pk}_1, c_1), \ldots, (\mathsf{pk}_w, c_w)\}$. This is so because x_1 is the only string which depended on $\mathbf{e}$, and $C(\mathbf{e})$ was considerate enough to provide it in (a). Coupled with the set $\mathsf{In} = \{(m_1, r_1), \ldots, (m_w, r_w)\}$ provided by the compressor in (b), D now has all the information to figure out the w input/output relationships missing in (c): namely, that $\mathbf{e}(\mathsf{pk}_i, m_i, r_i) = c_i$ for all i.

It remains to argue that under the conditions (i) of the Lemma, $C(\mathbf{e})$ would save more than $\log(1/\varepsilon)$ bits. Namely, the naive representation of w outputs compressed in parts (a)+(b) of the compression string is $3\lambda w$ bits. However, $C(\mathbf{e})$ only spent $|x_1| + w(1+\lambda)$ bits. Thus, we must have that:

$$|x_1| + w(1+\lambda) + \log(1/\varepsilon) \geq 3w\lambda$$

Recalling that we set $w = (\lceil 2\frac{|x_1|}{3\lambda} \rceil + 1)$, we get that $|x_1| \geq 3(w-1)\lambda/2$, and thus

$$3(w-1)\lambda/2 + w(1+\lambda) + \log(1/\varepsilon) \geq 3w\lambda$$

This easily implies $\log(1/\varepsilon) \geq w\lambda/2 \geq \lambda/2$, or $\varepsilon \leq 2^{-\lambda/2}$. □

Lemma A.3. *Let $X_1, \ldots, X_{t+1}$ be independent, Bernoulli random variables, where* $\Pr[X_i = 1] = p$, *for all $i \leq t+1$. Then*

$$\Pr[X_1 = 0 \wedge \cdots \wedge X_t = 0 \wedge X_{t+1} = 1] \leq \frac{1}{t}.$$

Definition A.4 (Heavy queries/responses)**.** *Suppose $\mathsf{A}^f(1^\lambda)$ is an oracle-aided algorithm with access to an oracle f. We say a query qu is p-heavy if the probability that qu is asked during a random execution of $\mathsf{A}^f(1^\lambda)$ is at least p. We say a response y is a p-heavy response if the probability that a query/answer of the form $(* \xrightarrow[f]{} y)$ occurs during a random execution of $\mathsf{A}^f(1^\lambda)$ is at least p. We say a set of query/answer pairs Freq contains a query qu if $(\mathsf{qu} \xrightarrow[f]{} *) \in \mathsf{Freq}$. We say a set of query/answer pairs Freq contains a response y if $(* \xrightarrow[f]{} y) \in \mathsf{Freq}$. Notice that if y has several pre-images (i.e., several $x_1, ..., x_w$ such that $f(x_i) = y$ for all $i \in [w]$), as long as Freq has at least one of those preimages (i.e., for some $i \in [w]$: $(x_i \xrightarrow[f]{} y) \in \mathsf{Freq}$), we say Freq contains the response y. That is, Freq does not need to contain all the pre-images of y to deem y contained in Freq.*

Lemma A.5 (Heavy query/response learner)**.** *Suppose $\mathsf{A}^f(1^\lambda)$ is an oracle-aided algorithm with access to an oracle f, making $t := t(\lambda) \in \mathsf{poly}(\lambda)$ queries. Suppose $p = \frac{1}{\mathsf{poly}(\lambda)}$, and let $\eta \geq \frac{\omega(\log \lambda)}{p}$. Let Freq be the set of all query/answer pairs generated during η random executions of $\mathsf{A}^f(1^\lambda)$. With probability at least $1 - \frac{1}{2^{\omega(\log(\lambda))}}$, the set Freq contains both all p-heavy queries qu and p-heavy responses y.*

Proof. We show that for any p-heavy query qu the probability that qu does not occur during η random executions is at most $\frac{1}{e^{\omega(\log \lambda)}}$. Similarly, we show that for any p-heavy response y, the probability that y never occurs as a response during η random executions is at most $\frac{1}{e^{\omega(\log \lambda)}}$. Since the number of p-heavy queries and p-heavy responses is at most $2t^2/p = \mathsf{poly}(\lambda)$ in total (see below on why), the probability of missing at least one p-heavy query qu or at least one p-heavy response y is at most $\frac{\mathsf{poly}(\lambda)}{e^{\omega(\log \lambda)}} \leq \frac{1}{2^{\omega(\log \lambda)}}$, as desired.

To see why we have at most $t^2/p = \mathsf{poly}(\lambda)$ p-heavy queries, note that if qu is p-heavy, then for some index $i \in [t]$: qu is ith p/t-heavy (i.e., the probability that the ith query is qu is at least p/t).

For any index $i \in [t]$ we have at most t/p queries which are ith p/t-heavy. Thus, we have at most $t^2/p = \mathsf{poly}(\lambda)$ queries which are globally p-heavy.

Similarly, if an output y is a p-heavy response, then, by definition, the probability that y occurs as a response to a query during a random execution of $\mathsf{A}^f(1^\lambda)$ is at least p. Thus, for some index $i \in [t]$: y is an ith p/t-heavy response; namely, the probability that the output of the ith query is y is at least p/t. For any $i \in [t]$ we have at most t/p responses which are ith p/t-heavy response. Thus, we have at most $t^2/p = \mathsf{poly}(\lambda)$ responses which are globally p-heavy.

For a p-heavy query qu and $i \in [\eta]$ let $x_i = 0$ if qu does not occur during the ith execution of $\mathsf{A}^f(1^\lambda)$. By Lemma A.1, the probability that qu does not appear during any of the η executions is at most $\frac{1}{e^{p\eta}} \leq \frac{1}{e^{\omega(\log \lambda)}}$.

Similarly, for a p-heavy response y, let $x_i = 0$ if y does not occur as a response during the ith execution of $\mathsf{A}^f(1^\lambda)$. By Lemma A.1, the probability that y never appears as a response during η executions is at most $\frac{1}{e^{p\eta}} \leq \frac{1}{e^{\omega(\log \lambda)}}$. □

Lemma A.6. *Let $\mathsf{A}^f(1^\lambda)$, t and $p = \frac{1}{\mathsf{poly}(\lambda)}$ be as in Lemma A.5. Let Freq be the set of all query/answer pairs generated during η random executions of $\mathsf{A}^f(1^\lambda)$, where $\eta \geq \frac{\omega(\log \lambda)}{p}$. Let Q be a set of query/answer pairs generated during a random execution of $\mathsf{A}^f(1^\lambda)$, made independently of Freq. Let L be a set of t queries made independently of Q. Then the probability that there exists a query in L which also appears in $\mathsf{Q} \setminus \mathsf{Freq}$ is at most $\frac{1}{2^{\omega(\log(\lambda))}} + tp$.*

Proof. Let Bad be the event we need to bound. We let $\mathsf{Collect}$ be the event that all p-heavy queries are collected by Freq. By Lemma A.5, $\Pr[\overline{\mathsf{Collect}}] \leq \frac{1}{2^{\omega(\log(\lambda))}}$. Assuming $\mathsf{Collect}$ holds, the probability that any fixed query of L appears in $\mathsf{Q} \setminus \mathsf{Freq}$ is at most p, and so the probability that some query of L appears in $\mathsf{Q} \setminus \mathsf{Freq}$ is at most tp. Thus,

$$\Pr[\mathsf{Bad}] \leq \Pr[\overline{\mathsf{Collect}}] + \Pr[\mathsf{Bad} \mid \mathsf{Collect}] \leq \frac{1}{2^{\omega(\log(\lambda))}} + tp, \tag{1}$$

as desired. □

Lemma A.7 (Intersection queries/responses learner)**.** *Let $\mathsf{A}^f(1^\lambda)$ be an oracle-aided algorithm making $t := t(\lambda) \in \mathsf{poly}(\lambda)$ queries. Suppose $p = \frac{1}{\mathsf{poly}(\lambda)}$, and let $\eta \geq \frac{\omega(\log \lambda)}{p}$. Let Freq be formed by recording all query/answer pairs made during η random executions of $\mathsf{A}^f(1^\lambda)$. Let Q_1 and Q_2 be the sets of query/answer pairs made during two random independent executions of $\mathsf{A}^f(1^\lambda)$. We say Q_1 and Q_2 have an intersection query if $(\mathsf{Q}_1 \cap \mathsf{Q}_2) \neq \emptyset$. We say Q_1 and Q_2 have an intersection response, if there exists some y such that y occurs as a response in both Q_1 and Q_2; namely, $(* \underset{f}{\rightarrow} y) \in \mathsf{Q}_1$ and $(* \underset{f}{\rightarrow} y) \in \mathsf{Q}_2$. The probability that there exists an intersection query or an intersection response between Q_1 and Q_2 which is not picked up by Freq is at most $2tp + \frac{1}{2^{\omega(\log \lambda)}}$. That is,*

$$\Pr[(((\mathsf{Q}_1 \cap \mathsf{Q}_2) \setminus \mathsf{Freq})) \neq \emptyset) \bigvee (\exists y \ s.t. \ (* \underset{f}{\rightarrow} y) \in \mathsf{Q}_1 \wedge (* \underset{f}{\rightarrow} y) \in \mathsf{Q}_2 \wedge (* \underset{f}{\rightarrow} y) \notin \mathsf{Freq})] \leq$$

$$2tp + \frac{1}{2^{\omega(\log \lambda)}}$$

Proof. Let $\mathsf{Intersect}$ be the probability of the event we want to bound. By Lemma A.5 the probability of the event, Pick, that all p-heavy queries qu and all p-heavy responses y are picked up by Freq

is at least $\alpha := 1 - \frac{1}{2^{\omega(\log\lambda)}}$. Assuming Pick, the probability that any fixed query qu of Q_1 appears in $\mathsf{Q}_2 \setminus \mathsf{Freq}$ is at most p. The reason is that since Pick holds and $\mathsf{qu} \notin \mathsf{Freq}$, the query qu is not p-heavy, hence occurring with probability at most p during another random execution. Similarly, the probability that any fixed response y contained in Q_1 also appears in $\mathsf{Q}_2 \setminus \mathsf{Freq}$ is at most p. Assuming Pick, the probability there exists $\mathsf{qu} \in \mathsf{Q}_1$ such that $\mathsf{qu} \in \mathsf{Q}_2 \setminus \mathsf{Freq}$ is at most tp, by the union bound. Similarly, assuming Pick, the probability there exists a response y in Q_1 such that y also appears in $\mathsf{Q}_2 \setminus \mathsf{Freq}$ is at most tp. Thus, $\Pr[\mathsf{Intersect}] \leq \frac{1}{2^{\omega(\log\lambda)}} + (1 - \frac{1}{2^{\omega(\log\lambda)}})2tp \leq 2tp + \frac{1}{2^{\omega(\log\lambda)}}$. The proof is now complete. □

Lemma A.8 (Hitting the image of random injective function)**.** *Let $\mathsf{A}^f(1^\lambda)$ be a polynomial-query algorithm with access to an oracle $f : \{0,1\}^\lambda \to \{0,1\}^{3\lambda}$ chosen uniformly at random from the set of all functions from $\{0,1\}^\lambda$ to $\{0,1\}^{3\lambda}$. The adversary A at the end outputs t points. We have*

$$\Pr[(y_1, \ldots, y_t) \leftarrow_{\$} \mathsf{A}^f(1^\lambda) \text{ and } \exists i \in [t] \text{ and } x \text{ s.t. } y_i = O(x) \wedge (* \underset{O}{\to} y_i) \notin \mathsf{Q}_{\mathsf{A}}] \leq 2^{-2\lambda},$$

where the probability is taken over the random choice of f as well as A's random coins, and where Q_{A} is the set of all A's query-answer pairs.

B Omitted Proofs from Section 4

Proof of Lemma 4.14. We prove this for $i = 1$. We will use Lemma A.6. For this proof, let Freq denote the value of Freq right after Line 2 (and before executing Line 3) of Brk's execution. If the event Empty_1 occurs, there must exist a query $\mathsf{qu} := (\mathsf{sk} \underset{\mathbf{g}}{\to} \mathsf{pk}) \in \mathsf{QGen}_1$ such that (a) $\mathsf{qu} \notin \mathsf{Freq}$ and $(\mathsf{sk} \underset{\mathbf{g}}{\to} *) \in \mathsf{Forbid}$, or (b) $(* \underset{\mathbf{g}}{\to} \mathsf{pk}) \in \mathsf{Forbid}$ and $(* \underset{\mathbf{g}}{\to} \mathsf{pk}) \notin \mathsf{Freq}$. If neither (a) nor (b) holds, there will always exist a $(\widetilde{\mathsf{SK}}_i, \mathbf{g}'_i)$ as per line 4a of Brk's computation. Let $p = \frac{\lambda^{0.1}}{\eta}$ and notice that $\eta \geq \frac{\omega(\log\lambda)}{p}$.

Define a set L of queries as follows:

- For any $(\mathsf{sk} \underset{\mathbf{g}}{\to} *) \in \mathsf{Forbid}$, add $(\mathsf{sk} \underset{\mathbf{g}}{\to} ?)$ to L; and
- for any $(* \underset{\mathbf{g}}{\to} \mathsf{pk}) \in \mathsf{Forbid}$, if $\mathbf{g}^{-1}(\mathsf{pk}) \neq \perp$, add $(\mathsf{sk} \underset{\mathbf{g}}{\to} ?)$ to L, where $\mathsf{sk} = \mathbf{g}^{-1}(\mathsf{pk})$.[14]

If the event Empty_1 occurs, then the event E defined as $\mathsf{Query}(\mathsf{QGen}_1 \setminus \mathsf{Freq}) \cap \mathsf{L} \neq \emptyset$ holds.

Since the set Forbid, as well as L, are formed independently of QGen_1, and that L has most $2\eta'\lambda$ elements, invoking Lemma A.6 for $p = \frac{\lambda^{0.1}}{\eta}$ and $t := |\mathsf{L}| \leq 2\eta'\lambda$

$$\Pr[\mathsf{Empty}_1] \leq \Pr[E] \leq \frac{1}{2^{\omega(\log\lambda)}} + \frac{2\lambda^{1.1}\eta'}{\eta}. \tag{2}$$

□

Proof of Lemma 4.15. We prove this for a fixed value of h (say, $h = 1$), and the overall bound will follow via a union bound, via an additional multiplicative factor of n. Recall that Agree is the event that the union set $\mathsf{S} := \mathsf{QC} \cup \mathsf{QEnc} \cup_{i \neq 1} \mathsf{QGen}_i$ agrees with $\mathbf{g}'_1$, where QEnc, QC and QGen_i

[14] Since $\mathbf{g}$ is chosen at random and has a sparse range, there will exist at most one pre-image, with all but negligible probability.

are defined in Notation 4.12. First, notice that $\mathbf{g}'_1$ always agrees with QEnc, because the former has only $\mathbf{g}$-type queries, while the latter has $\mathbf{e}$-type queries. Thus, we need to bound the probability that for any $(x \underset{g}{\rightarrow} y) \in \mathsf{QC} \cup_{i\neq 1} \mathsf{QGen}_i$, either $(x \underset{g}{\rightarrow} *) \notin \mathbf{g}'_1$ or $(x \underset{g}{\rightarrow} y) \in \mathbf{g}'_1$.

We break up the event Agree into Agree_1 and Agree_2. We let Agree_1 be the event QC agrees with $\mathbf{g}'_1$, and let Agree_2 be the event $\mathsf{S}' := \cup_{i\neq 1}\mathsf{QGen}_i$ agrees with $\mathbf{g}'_1$.

We first bound the probability of $\overline{\mathsf{Agree}_1}$. Let P be the process performed in each iteration of Line 3 of Brk, namely the process of sampling a fresh $(\mathsf{PK}, *)$ and running $(\widetilde{\mathsf{SK}}, \mathbf{g}') \leftarrow_{\$} \mathsf{ConsOrc}(\mathsf{PK}, \mathsf{Freq}, \mathsf{Forbid})$, and updating Forbid accordingly. Notice that $(\widetilde{\mathsf{SK}}_1, \mathbf{g}'_1)$ of Line 4a of Brk is sampled according to the same process. We say an iteration of the process P is Good if either (a) the sampled $\mathbf{g}'$ in that iteration is empty; or (b) $\mathsf{Query}(\mathbf{g}') \cap \mathsf{Query}(\mathsf{QC} \setminus \mathsf{Freq}) = \emptyset$. Otherwise, we say that iteration of P is Bad. The event $\overline{\mathsf{Agree}_1}$ is the event that the iteration corresponding to $(\widetilde{\mathsf{SK}}_1, \mathbf{g}'_1)$ in Line 4a is Bad. Also, notice that since $|\mathsf{QC}| = \lambda$, we have at most λ Bad iterations. The reason for this is that if for some iteration the event Bad happens, then the particular query of QC which caused Bad is added to Forbid, and so the same query cannot make Bad happen in a future iteration. Now since the iteration for $(\widetilde{\mathsf{SK}}_1, \mathbf{g}')$ is the $(\gamma+1)$'s iteration, where $\gamma \leftarrow_{\$} [\eta']$, the probability that that iteration is Bad is at most $\frac{\lambda}{\eta'}$. Thus, $\Pr[\overline{\mathsf{Agree}_1}] \leq \frac{\lambda}{\eta'}$.

We now bound the event $\overline{\mathsf{Agree}_2}$. Notice if $\overline{\mathsf{Agree}_1}$ holds, there must exist a query in $\mathbf{g}'_1$ that also appears in $\mathsf{QGen}_i \setminus \mathsf{Freq}$ for some $i > 1$. Let $p = \frac{\lambda^{0.1}}{\eta}$ and notice that $\eta \geq \frac{\omega(\log p)}{p}$. Since QGen_i for all $i \in \{2, \ldots, n\}$ is formed independently of $\mathbf{g}'_1$, applying Lemma A.6 for p and η, the probability there is an intersection between $\mathbf{g}'_1$ and $(\cup_{i\neq 1}\mathsf{QGen}_i) \setminus \mathsf{Freq}$ is at most $\frac{1}{2^{\omega(\log(\lambda))}} + \frac{n\lambda^{1.1}}{\eta}$. Here we used the fact that $|\cup_{i\neq 1} \mathsf{QGen}_i| < n\lambda$.

□

Proof of Lemma 4.16. We prove it for $i = 1$. The proof works exactly as that of bounding $\Pr[\mathsf{Agree}_1]$ in Lemma 4.15. So, we repeat the argument with the necessary modifications. Let P be the process performed in each iteration of Line 3 of Brk, namely the process of sampling a fresh $(\mathsf{PK}, *)$ and running $(\widetilde{\mathsf{SK}}, \mathbf{g}') \leftarrow_{\$} \mathsf{ConsOrc}(\mathsf{PK}, \mathsf{Freq}, \mathsf{Forbid})$, and updating Forbid accordingly. Notice that $(\widetilde{\mathsf{SK}}_1, \mathbf{g}'_1)$ of Line 4a of Brk is sampled according to the same process. We say an iteration of the process P is Good if either (a) the sampled $\mathbf{g}'$ in that iteration is empty; or (b) there does not exist a query/answer pair $(* \underset{\mathbf{g}}{\rightarrow} \mathsf{pk}) \in \mathbf{g}'$ such that $(* \underset{\mathbf{g}}{\rightarrow} \mathsf{pk}) \in \mathsf{QA} \setminus \mathsf{Freq}$. Otherwise, we say that iteration of P is Bad. The event $\overline{\mathsf{Surprise}_1}$ is the event that the iteration corresponding to $(\widetilde{\mathsf{SK}}_1, \mathbf{g}'_1)$ in Line 4a is Bad. Also, notice that since $|\mathsf{QC}| = \lambda$, we have at most λ Bad processes. The reason for this is that if for some iteration the event Bad happens, then the particular public key pk of QC which caused Bad is added to Forbid, and so the same pk cannot make Bad happen in a future iteration. Now since the iteration for $(\widetilde{\mathsf{SK}}_1, \mathbf{g}')$ is the $(\gamma + 1)$'s iteration, where $\gamma \leftarrow_{\$} [\eta']$, the probability that that iteration is Bad is at most $\frac{\lambda}{\eta'}$. □

Proof of Lemma 4.17. We show whenever the event Spoof holds, we can forge a public key pk in the sense of Lemma A.8, implying the bound of $\frac{1}{2^{2\lambda}}$. We build an adversary A in the sense of Lemma A.8, as follows. The adversary A will generate all $(\mathsf{PK}_1, \ldots, \mathsf{PK}_n)$, which are the input to Brk, by running $\mathsf{CRS} \leftarrow_{\$} \mathrm{CRSGen}^{\mathbf{g}}(1^\lambda)$ and $(\mathsf{PK}_i, *) \leftarrow_{\$} \mathrm{Init}^{\mathbf{g}}(\mathsf{CRS})$. Then A performs the steps of Brk up until producing $\mathbf{g}'_1, \ldots, \mathbf{g}'_n$ – while populating Freq as in Brk's procedure. At this point notice that all queries made to $\mathbf{g}$ by A thus far are either contained in $\mathsf{QA} \cup \mathsf{QGen}_1 \cup \cdots \cup \mathsf{QGen}_n$ or in Freq. Thus, if the event Spoof holds, then A has indeed forged a pk. □

Proof of Lemma 4.19. We claim $\Pr[\mathsf{Evnt}_2 \wedge \overline{\mathsf{Surprise}} \wedge \overline{\mathsf{Spoof}} \wedge \overline{\mathsf{Intersect}}] \leq \mathsf{negl}(\lambda)$. Assuming this, by Lemmas 4.16, 4.17, 4.14, 4.18

$$\Pr[\mathsf{Surprise} \vee \mathsf{Spoof} \vee \mathsf{Intersect}] \leq \frac{n\lambda}{\eta'} + \frac{1}{2^{2\lambda}} + \frac{2n^2\lambda^{1.1}}{\eta} + \frac{n^2}{2^{\omega(\log \lambda)}}. \tag{3}$$

Assuming $\eta' \geq n\lambda^{c+1}$ and $\eta \geq n\eta'\lambda^{1.1+c}$,

$$\Pr[\mathsf{Surprise} \vee \mathsf{Spoof} \vee \mathsf{Intersect}] \leq \frac{4}{\lambda^c}. \tag{4}$$

Thus, $\Pr[\mathsf{Evnt}_2] \leq \frac{5}{\lambda^c}$, as desired.

To prove the above claim, we show whenever all the events

$$\mathsf{Evnt}_2 \wedge \overline{\mathsf{Surprise}} \wedge \overline{\mathsf{Spoof}} \wedge \overline{\mathsf{Intersect}}$$

hold, we can build a forger in the sense of Lemma A.2. The desired bound will then follow.

Since Evnt_2 holds, $\mathsf{Forge} = 1$. Let $\mathsf{Chal} := \{(\mathsf{pk}_1, c_2), \ldots, (\mathsf{pk}_n, c_n)\}$ be the set of public key/ciphertexts built up by Brk. To apply Lemma A.2, think of $(\mathsf{PK}_1, \ldots, \mathsf{PK}_n)$ as x_0, of C as x_1, and of Chal as the challenge set required by the lemma. First, note that a forger $\mathsf{B}^{\mathbf{g},\mathbf{u},\mathbf{v},\mathbf{d}}(x_0, x_1)$ may indeed efficiently compute Chal, because Brk never makes $\mathbf{e}$ queries, so Brk can be simulated by B. We have to ensure: (i) $n \geq \lceil 2\frac{|C|}{3\lambda} \rceil + 1$; (ii) all the pairs in Chal are distinct, and (iii) for all $i \in [n]$ $\mathbf{v}(\mathsf{pk}_i, c_i) = \top$. Condition (i) holds because $|C| \leq \frac{3\lambda(n-1)}{2}$, by assumption. Condition (iii) holds by the check (4(c)i) made by Brk.

We now show Condition (ii) holds. We will prove a stronger statement by showing that in fact all pk_i's in the set Chal are distinct.[15] For any $i \in [n]$, we claim: (a) $(* \xrightarrow[\mathbf{g}]{} \mathsf{pk}_i) \in \mathbf{g}'_i$, (b) $(* \xrightarrow[\mathbf{g}]{} \mathsf{pk}_i) \notin \mathsf{Freq}$, and (c) $\mathsf{pk}_i \in \mathbf{g}(*)$. Conditions (a) and (b) hold because otherwise none of the sub-bullets of Line 4c (and in particular Line 4(c)iii) would be hit, and so (pk_i, c_i) would not be added to Chal. Also, (c) holds because otherwise Line (4(c)ii) of Brk would be hit, and hence (pk_i, c_i) would not be added to Chal.

Now since $\overline{\mathsf{Spoof}}$ holds, Condition (a) and (c) imply that for some sk_i, $(\mathsf{sk}_i \xrightarrow[\mathbf{g}]{} \mathsf{pk}_i) \in \cup_{i\in[n]}\mathsf{QGen}_i \cup \mathsf{QC} \cup \mathsf{Freq}$. Since (b) holds, $(* \xrightarrow[\mathbf{g}]{} \mathsf{pk}_i) \in \cup_{i\in[n]}\mathsf{QGen}_i \cup \mathsf{QC}$. Since $\overline{\mathsf{Surprise}}$ and (b) hold, $(* \xrightarrow[\mathbf{g}]{} \mathsf{pk}_i) \in \cup_{i\in[n]}\mathsf{QGen}_i$. Finally, since (b) and $\overline{\mathsf{Intersect}}$ hold, for all distinct i and j: $\mathsf{pk}_i \neq \mathsf{pk}_j$. Indeed, since we already know $(* \xrightarrow[\mathbf{g}]{} \mathsf{pk}_i) \in \mathsf{QGen}_i \setminus \mathsf{Freq}$ and $(* \xrightarrow[\mathbf{g}]{} \mathsf{pk}_j) \in \mathsf{QGen}_j \setminus \mathsf{Freq}$, if $\mathsf{pk}_i = \mathsf{pk}_j$, then the assumption that $\overline{\mathsf{Intersect}}$ holds will be violated. The proof is now complete.

□

C CKE Impossibility from Black-Box PKE: General Case

In this section, we present the general attack against CKE constructions that make use of the oracles in arbitrary ways. To make proofs simpler, we assume that the CKE protocol is in a *complied form*, with oracle access as $(\mathrm{CRSGen}^{\mathbf{g},\mathbf{e}}, \mathrm{Init}^{\mathbf{g},\mathbf{e}}, \mathrm{Comm}^{\mathbf{g},\mathbf{e}}, \mathrm{Derive}^{\mathbf{g},\mathbf{e},\mathbf{d}})$. Namely, we assume that no calls to $\mathbf{d}$ are made by CRSGen, Init and Comm. One may put any protocol into this compiled form by using standard compilation techniques [19, 26]. First, notice that we might assume that CRSGen

[15] In order for Lemma A.2 to apply it suffices to prove that the pairs are distinct.

makes no $\mathbf{d}$ queries, because answers to such queries can be predicted. Next, for Init we might assume that it does not make any $\mathbf{d}$ queries that it knows the answers to already (i.e., any query $((\mathsf{sk}, c) \underset{\mathbf{d}}{\rightarrow} ?)$ such that Init has already produced a query/response pair $((\mathsf{pk}, *, *) \underset{\mathbf{e}}{\rightarrow} c)$, where $\mathsf{pk} := \mathbf{g}(\mathsf{sk})$). Thus, the only non-trivial decryption queries are of the form $((\mathsf{sk}, c) \underset{\mathbf{d}}{\rightarrow} ?)$ where c was generated by $\mathsf{CRS} \leftarrow_\$ \mathrm{CRSGen}(1^\lambda)$. Here is where the idea of compilations comes into play: we might compile CRSGen and Init such that CRSGen will sample many random executions of Init in its head and will collect all decryption queries appearing there; then, it will append the answers to all such queries into CRS. This way we can change Init so that it will make no decryption queries. Similarly, we can get rid of $\mathbf{d}$ queries for Comm, since both preceding algorithms (namely, CRSGen and Init) can run many executions of Comm and provide decryption answers as part of their local outputs. Finally, we note that the reason we cannot do this step of ridding of $\mathbf{d}$ queries as easily for Derive is that Derive takes in a private key, which is not available to the previous algorithms.

For our general case we need a more general version of the output compression lemma, as given below. Informally, under this more general version, we allow the forger to call $\mathbf{e}$, but it should forge public key/ciphertexts pairs that are not generated as part of $\mathbf{e}$ queries. Due to this restriction, the proof template of Lemma A.2 still holds in this more general case, and we get the following more general variant (whose proof is omitted).

Lemma C.1. *Let $\mathsf{A}^{\mathbf{g},\mathbf{e}}$ be an arbitrary adversary that makes a number of queries and outputs an 'advice' string x. Let $\mathsf{B}^{\mathbf{g},\mathbf{e},\mathbf{u},\mathbf{v},\mathbf{d}}(x)$ be an adversary that takes as input x, makes queries to $(\mathbf{g}, \mathbf{e}, \mathbf{u}, \mathbf{v}, \mathbf{d})$ and outputs a set $\mathsf{Chal} = \{(\mathsf{pk}_1, c_1), \ldots, (\mathsf{pk}_w, c_w)\}$. Suppose Q is the set of all queries/responses made by B. We say Chal is non-trivial if for no $i \in [w]$, $((\mathsf{pk}_i, *, *) \underset{\mathbf{e}}{\rightarrow} c_i) \in \mathsf{Q}$. We say the event $\mathsf{Success}$ holds if (i) $w \geq \lceil 2^{\frac{|x|}{3\lambda}} \rceil + 1$; (ii) all the pairs in Chal are distinct, (iii) for all $i \in [w]$ $\mathbf{v}(\mathsf{pk}_i, c_i) = \top$ and (iv) Chal is non-trivial. We then have $\Pr[\mathsf{Success}] = \mathsf{negl}(\lambda)$, where the probability is taken over $(\mathbf{g}, \mathbf{e}, \mathbf{d}, \mathbf{u}, \mathbf{v}) \leftarrow_\$ \Psi$ and the random coins of A and B.*

For the general attack, we need to enhance the frequent queries learner, so that it also learns the heavy queries of Comm, in addition to those of Init.

Definition C.2 (Sampling frequent queries)**.** *We define a probabilistic oracle procedure $\mathsf{FreqPub}^{\mathbf{O}}$:*

- ***Input:*** *(CRS, η), where η is an integer.*
- ***Output:*** *A set of query/response pairs Freq sampled as follows. Let $\mathsf{Freq} = \emptyset$. Do the following η times, and record all query/answer pairs in Freq.*
    1. *Sample n public keys $\mathsf{PK}_1, \ldots, \mathsf{PK}_n$ by running $\mathrm{Init}^{\mathbf{g},\mathbf{e}}(\mathsf{CRS})$ n different times.*
    2. *Execute $\mathrm{Comm}^{\mathbf{g},\mathbf{e}}(\mathsf{PK}_1, \ldots, \mathsf{PK}_n)$.*

We also need a procedure that allows us to super-impose a set of encryption queries Q_c (sampled independently of $\mathbf{e}$) into $\mathbf{e}$, in such a way that the super-imposed encryption oracle, $\mathbf{e}_{\mathsf{imp}}$, agrees with Q_c, and with $\mathbf{e}$ as much as possible, and also that $\mathbf{e}_{\mathsf{imp}}$ has a corresponding super-imposed decryption oracle.

Definition C.3. *Let $\mathbf{O} := (\mathbf{g}, \mathbf{e}, \mathbf{d})$ be a Ψ-valid oracle and let*

$$\mathsf{Q}_c := \{\{((\mathsf{pk}_1, b_1, r_1) \underset{\mathbf{e}}{\rightarrow} c_1), \ldots, ((\mathsf{pk}_w, b_w, r_w) \underset{\mathbf{e}}{\rightarrow} c_w)\}\}$$

be a set of $\mathbf{e}$*-type query answer pairs, which may not agree with* $\mathbf{e}$*. We define* $(\mathbf{e}_{\mathsf{imp}}, \mathbf{d}_{\mathsf{imp}}) := \mathsf{Q}_c \Diamond^* \mathbf{O}$*, obtained by super-imposing* Q_c *on* $\mathbf{O}$*, as follows.*

First, let $\mathsf{W} = \{(\mathsf{pk}_1, c_1), \ldots, (\mathsf{pk}_p, c_p)\}$ *and*

$$\mathsf{W}' = \{(\mathsf{pk}_1, \mathbf{e}(pk_1, b_1, r_1)), \ldots, (\mathsf{pk}_p, \mathbf{e}(pk_p, b_p, r_p))\}.$$

Define

$$\mathbf{e}_{\mathsf{imp}}(pk, b, r) = \begin{cases} c_i & \textit{if } (\mathsf{pk}, b, r) = (\mathsf{pk}_i, b_i, r_i), \textit{ for some } i \in [w] \\ \hat{c} & \textit{else if } (\mathsf{pk}, \mathbf{e}(pk, b, r)) \in \mathsf{W}, \\ \mathbf{e}(pk, b, r) & \textit{otherwise} \end{cases} \tag{5}$$

where $\hat{c}$ *is defined as follows: Letting* x *be the smallest integer such that* $(pk, \mathbf{e}(pk, b, r+x)) \notin \mathsf{W} \cup \mathsf{W}'$ *we set* $\hat{c} = \mathbf{e}(pk, b, r+x)$*. Here,* $r + x$ *is done using a standard method.*

$$\mathbf{d}_{\mathsf{imp}}(\mathsf{sk}, c) = \begin{cases} b_i & \textit{if } \mathbf{g}(\mathsf{sk}) = \mathsf{pk}_i \textit{ and } c = c_i \textit{ for some } 1 \leq i \leq w \\ \mathbf{d}(\mathsf{sk}, c) & \textit{otherwise} \end{cases} \tag{6}$$

C.0.1 Description of the Attacker

We now give the CKE attacker for the general case. The attacker will at the end output a (polynomial-sized) set of keys, and we will be interested in the probability that the set contains the shared key. Since the set has a polynomial number of keys, we can also guess the correct key with a non-negligible probability, assuming that the set has indeed the correct key.

$\mathsf{Brk}^{\mathbf{g},\mathbf{e},\mathbf{d},\mathbf{u},\mathbf{v}}(\mathsf{CRS}, \mathsf{PK}_1, \ldots, \mathsf{PK}_n, C)$:

1. Let $\mathsf{DecValues} := \emptyset$ and $\mathsf{Forbid} = \emptyset$. The set $\mathsf{DecValues}$ will maintain all decryption values.

2. Do the following for η iterations. Sample randomness R and execute $\mathsf{FreqPub}^{\mathbf{O}}(\mathsf{CRS})$ and record all query/response pairs in Freq.

3. Sample $\gamma \leftarrow_{\$} [\eta']$ and do the following γ times. Run $(\mathsf{PK}, *) \leftarrow_{\$} \mathrm{Init}^{\mathbf{g}}(\mathsf{CRS})$ using fresh randomness, sample $(\widetilde{\mathsf{SK}}, \mathbf{g}', \mathbf{e}') \leftarrow_{\$} \mathsf{ConsOrc}(\mathsf{PK}, \mathsf{Freq}, \mathsf{Forbid})$, and

   (a) for every $\mathsf{qu} := (\mathsf{sk} \underset{\mathbf{g}}{\rightarrow} \mathsf{pk}) \in \mathbf{g}' \setminus \mathsf{Freq}$, add $(\mathsf{sk} \underset{\mathbf{g}}{\rightarrow} ?)$ to Forbid. Also, for any $(\mathsf{sk} \underset{\mathbf{g}}{\rightarrow} \mathsf{pk}) \in \mathbf{g}' \setminus \mathsf{Freq}$ if $(* \underset{\mathbf{g}}{\rightarrow} \mathsf{pk}) \notin \mathsf{Freq}$, add $(* \underset{\mathbf{g}}{\rightarrow} \mathsf{pk})$ to Forbid.

   (b) for every $\mathsf{qu} := ((\mathsf{pk}, b, r) \underset{\mathbf{e}}{\rightarrow} c) \in \mathbf{e}' \setminus \mathsf{Freq}$, add $((\mathsf{pk}, b, r) \underset{\mathbf{e}}{\rightarrow} ?)$ to Forbid. Also, for $((\mathsf{pk}, b, r) \underset{\mathbf{e}}{\rightarrow} c) \in \mathbf{e}' \setminus \mathsf{Freq}$, if $((\mathsf{pk}, *, *) \underset{\mathbf{e}}{\rightarrow} c) \notin \mathsf{Freq}$, add $((\mathsf{pk}, *, *) \underset{\mathbf{e}}{\rightarrow} c)$ to Forbid.

   At the end of each iteration, update Freq by adding all queries made during $(\mathsf{PK}, *) \leftarrow_{\$} \mathrm{Init}^{\mathbf{g}}(\mathsf{CRS})$ to Freq.

4. For $i \in [n]$

   (a) Sample $(\widetilde{\mathsf{SK}}_i, \mathbf{g}'_i, \mathbf{e}'_i) \leftarrow_{\$} \mathsf{ConsOrc}(\mathsf{PK}_i, \mathsf{Freq}, \mathsf{Forbid})$. If $(\widetilde{\mathsf{SK}}_i, \mathbf{g}'_i, \mathbf{e}'_i) = \perp$, then halt.

   (b) Let $\mathbf{O}''_i = (\mathbf{g}, \mathbf{e}''_i, \mathbf{d}''_i) := \mathbf{e}'_i \Diamond^* \mathbf{O}$.

   (c) Let $(\widetilde{\mathbf{g}}_i, \widetilde{\mathbf{e}}_i, \widetilde{\mathbf{d}}_i) := \mathbf{g}'_i \Diamond^* \mathbf{O}''_i$.

(d) Parse $\mathbf{e}'_i := \{((\mathsf{pk}_1, *, *) \xrightarrow[\mathbf{e}]{} c_1), \ldots, ((\mathsf{pk}_\lambda, *, *) \xrightarrow[\mathbf{e}]{} c_\lambda)\}$. For all $j \in [\lambda]$, add both (pk_i, c_i) and $(\mathsf{pk}_i, \mathbf{e}(\mathsf{pk}_i, b_i, r_i))$ to Q. Also, for any (pk, c) such that $((\mathsf{pk}, *, *) \xrightarrow[\mathbf{e}]{} c) \in \mathsf{Freq}$, add (pk, c) to Q.

(e) Execute $\mathrm{Derive}^{\widetilde{\mathbf{g}}_i, \widetilde{\mathbf{e}}_i, \widetilde{\mathbf{d}}_i}(\widetilde{\mathsf{SK}}_i, C)$ and reply to queries as follows. Reply to all $\widetilde{\mathbf{g}}_i$ and $\widetilde{\mathbf{e}}_i$ queries as Part 2 of Lemma C.4. For a query $\mathsf{qu} := ((\mathsf{sk}, c) \xrightarrow[\widetilde{\mathbf{d}}_i]{} ?)$, if $(\mathsf{sk} \xrightarrow[\mathbf{g}]{} *) \in \mathsf{Freq}$ or $(\mathsf{sk} \xrightarrow[\mathbf{g}]{} *) \notin \mathbf{g}'$, then reply to the query as in Part 3 of Lemma C.4. Otherwise, let $\mathsf{pk} = \widetilde{\mathbf{g}}_i(\mathsf{sk})$ — which can be computed efficiently — and

   i. if $(\mathsf{sk}' \xrightarrow[\mathbf{g}]{} \mathsf{pk}) \in \mathsf{Freq}$ for some sk', reply to qu with $\mathbf{d}(\mathsf{sk}', c)$;

   ii. if $(\mathsf{pk}, c) \in \mathsf{Q}$, then reply to the query as in Part 4 of Lemma C.4.

   iii. else if $\mathbf{v}(\mathsf{pk}, c) = \perp$, then reply to qu with $\perp$;

   iv. else if $\mathbf{u}(\mathsf{pk}, c) = m \neq \perp$, then reply to qu with m;

   v. else, reply to qu with $\perp$ and add (pk, c) to Chal.

(f) Letting $\widetilde{K}_i$ be the output of the simulated decryption of $\mathrm{Derive}^{\widetilde{\mathbf{d}}}(\widetilde{\mathsf{SK}}_i, C)$, add $\widetilde{K}_i$ to $\mathsf{DecValues}$.

We give the following lemma whose proof is immediately obtained via inspection, and so the proof is omitted.

Lemma C.4. *Fix an Iteration $i \in [n]$ in Step 4 of Brk's procedure. Let $\mathbf{e}'_i := \{((\mathsf{pk}_j, b_j, r_j) \xrightarrow[\mathbf{e}]{} c_j) \mid j \in [\lambda]\}$. Define*

$$\mathsf{W} := \{(\mathsf{pk}_j, c_j) \mid j \in [\lambda]\} \tag{7}$$

$$\mathsf{W}' := \{(\mathsf{pk}_j, \mathbf{e}(\mathsf{pk}_j, b_j, r_j) \mid j \in [n]). \tag{8}$$

Let $\mathsf{Q} := \mathsf{W} \cup \mathsf{W}' \cup \{(\mathsf{pk}, c) \mid ((\mathsf{pk}, *, *) \xrightarrow[\mathbf{e}]{} c) \in \mathsf{Freq}\}$, as in Step 4d of Brk's procedure.

1. *$(\widetilde{\mathbf{g}}_i, \widetilde{\mathbf{e}}_i, \widetilde{\mathbf{d}}_i)$ is a PKE-valid oracle; i.e., it satisfies perfect correctness (Definition 4.7).*

2. *Both $\widetilde{\mathbf{g}}_i$ and $\widetilde{\mathbf{e}}_i$ (Step 4c) can be computed efficiently on all points by having oracle access to $(\mathbf{g}, \mathbf{e})$ and having $(\mathbf{g}'_i, \mathbf{e}'_i)$ as input.*

3. *For any (sk, c) if $(\mathsf{sk} \xrightarrow[\mathbf{g}'_i]{} ?) \notin \mathbf{g}'_i$, or $(\mathsf{sk} \xrightarrow[\mathbf{g}]{} ?) \in \mathsf{Freq}$, or $(* \xrightarrow[\mathbf{g}]{} \mathsf{pk}) \in \mathsf{Freq}$ where $\mathsf{pk} := \widetilde{\mathbf{g}}_i(\mathsf{sk})$, then $\tilde{\mathbf{d}}(\mathsf{sk}, c)$ can be efficiently computed by having oracle access to $(\mathbf{g}, \mathbf{e}, \mathbf{d})$ and having $\mathbf{g}'_i$ and $\mathbf{e}'_i$ as an input.*

4. *For any query $((\mathsf{sk}, c) \xrightarrow[\widetilde{\mathbf{d}}]{} ?)$, letting $\mathsf{pk} := \widetilde{\mathbf{g}}_i(\mathsf{sk})$ (which can be computed efficiently as per Line 2), if $(\mathsf{pk}, c) \in \mathsf{Q}$, then $((\mathsf{sk}, c) \xrightarrow[\widetilde{\mathbf{d}}]{} ?)$ can be computed efficiently, as follows. If $((\mathsf{pk}, b, r) \xrightarrow[\mathbf{e}]{} c) \in \mathsf{Freq}$ for some b and c, then $\widetilde{\mathbf{d}}(\mathsf{sk}, c) = b$. If $(\mathsf{pk}, c) \in \mathsf{W}' \setminus \mathsf{W}$, then $\widetilde{\mathbf{d}}(\mathsf{sk}, c) = \perp$. Else, if $(\mathsf{pk}, c) = (\mathsf{pk}_j, c_j)$ for $j \in [\lambda]$, then $\widetilde{\mathbf{d}}(\mathsf{sk}, c) = b_j$.*

C.0.2 Analysis of the Attack

We work with the events given in Definition 4.13 with the following modifications. We define the event

- Event Evnt_2: the event that for every $i \in [n]$, Line 4(e)v is hit with a value (pk, c) such that $((\mathsf{pk}, *, *) \underset{\mathbf{e}}{\to} c) \in \mathsf{QEnc} \setminus (\cup_i \mathsf{QGen_i} \cup \mathsf{QC} \cup \mathsf{Freq})$.

We will bound all these bounds below. The bounds for events $\mathsf{Surprise}$, Spoof and $\mathsf{Intersect}$ are exactly those of Lemmas 4.16, 4.17 and 4.18, with exactly the same proofs. We now bound the other events which require a slightly more general analysis.

Lemma C.5. *Assuming $\eta \geq \lambda^{0.1}$, for any $i \in [n]$* $\Pr[\mathsf{Empty}_i] \leq \frac{1}{2^{\omega(\log \lambda)}} + \frac{\lambda^{1.1}\eta'}{\eta}$. *Thus,* $\Pr[\mathsf{Empty}] \leq \frac{n}{2^{\omega(\log \lambda)}} + \frac{n\lambda^{1.1}\eta'}{\eta}$

Proof. We prove this for $i = 1$. The event Empty_1 occurs if there exists a query $\mathsf{qu} \in \mathsf{QGen}_1$ such that $\in \mathsf{Forbid} \setminus \mathsf{Freq}$. This is so because QGen_1 consists only of $\mathbf{g}$ and $\mathbf{e}$-type queries, and as long as all queries of QGen_1 can be picked by $\mathbf{g}'_1$, the event Empty_1 will not occur. But in order for a $\mathbf{g}$ and $\mathbf{e}$-type query of QGen_1 becomes “off-limits”, the same query should have been put in Forbid. Now exactly as in Lemma 4.14 we may conclude $\Pr[\mathsf{Empty}_i] \leq \frac{1}{2^{\omega(\log \lambda)}} + \frac{\lambda^{1.1}\eta'}{\eta}$. □

Lemma C.6. *Assuming $\eta \geq \lambda^{0.1}$,* $\Pr[\mathsf{Agree}] \geq 1 - \frac{3n}{2^{\omega(\log(\lambda))}} - \frac{(n^2+n)\lambda^{1.1}}{\eta} - \frac{3\lambda n}{\eta'} - \frac{\lambda^{1.1}}{\eta}$.

Proof. We prove this for a fixed value of h (say, $h = 1$), and the overall bound will follow via a union bound. We let Agree' be the event that g'_1 agrees with $\mathsf{QC} \cup \mathsf{QEnc} \cup_{i \neq 1} \mathsf{QGen}_i$ and Agree'' be the event that e'_1 agrees with $\mathsf{QC} \cup \mathsf{QEnc} \cup_{i \neq 1} \mathsf{QGen}_i$. It is easy to see that the probability of $\overline{\mathsf{Agree}}$ is at most $\Pr[\overline{\mathsf{Agree}'}] + \Pr[\overline{\mathsf{Agree}''}]$. Using the same argument as in Lemma 4.15, $\Pr[\overline{\mathsf{Agree}'}] \leq \frac{1}{2^{\omega(\log(\lambda))}} + \frac{\lambda}{\eta'} + \frac{n\lambda^{1.1}}{\eta}$.

To bound Agree'' we break up Agree'' into Agree'', Agree''_2 and Agree''_3, where these describe the events that $\widetilde{\mathbf{e}}_1$ agrees with QC, with $\cup_{i \neq 1}\mathsf{QGen}_i$, and with QEnc, respectively.

We first bound Agree''_1. Let P be the process performed in each iteration of Line 3 of Brk, namely the process of sampling a fresh $(\mathsf{PK}, *)$ and running $(\mathsf{SK}', \mathbf{g}', \mathbf{e}') \leftarrow_{\$} \mathsf{ConsOrc}(\mathsf{PK}, \mathsf{Freq}, \mathsf{Forbid})$, and updating Forbid accordingly. Notice that $(\widetilde{\mathsf{SK}}_1, \mathbf{g}'_1, \mathbf{e}'_1)$ of Line 4a of Brk is sampled according to the same process. We say an iteration of the process P is Good if either (a) the sampled $\mathbf{e}'$ in that iteration is empty; or (b) there does not exists any $((\mathsf{pk}, b, r) \underset{\mathbf{e}}{\to} c) \in \mathsf{QC} \setminus \mathsf{Freq}$ such that either $((\mathsf{pk}, b, r) \underset{\mathbf{e}}{\to} *) \in \mathbf{e}' \setminus \mathsf{Freq}$ or $((\mathsf{pk}, *, *) \underset{\mathbf{e}}{\to} c) \in \mathbf{e}' \setminus \mathsf{Freq}$. Otherwise, we say that iteration of P is Bad. By inspection, one can see that whenever $\overline{\mathsf{Agree}''_1}$ holds, the iteration corresponding to $(\widetilde{\mathsf{SK}}_1, \mathbf{g}'_1, \mathbf{e}'_1)$ in Line 4a is Bad. Also, notice that since $|\mathsf{QC}| = \lambda$, we have at most 2λ Bad iterations. This is so because any query in QC can make at most two iterations Bad. Now since the iteration for $(\widetilde{\mathsf{SK}}_1, \mathbf{g}'_1, \mathbf{e}'_1)$ is the $(\gamma + 1)$’s iteration, where $\gamma \leftarrow_{\$} [\eta']$, the probability that that iteration is Bad is at most $\frac{2\lambda}{\eta'}$. Thus, $\Pr[\overline{\mathsf{Agree}_2}] \leq \frac{2\lambda}{\eta'}$.

Exactly as in the proof of Lemma 4.15 we can deduce $\Pr[\mathsf{Agree}''_2] \leq \frac{1}{2^{\omega(\log(\lambda))}} + \frac{n\lambda^{1.1}}{\eta}$.

To bound Agree''_3, notice if Agree''_3 holds, there must exist a query in $\widetilde{\mathbf{e}}_1$ that also appears in QEnc. Let $p = \frac{\lambda^{0.1}}{\eta}$ and notice that $\eta \geq \frac{\omega(\log p)}{p}$. Since QEnc is formed independently of $\widetilde{\mathbf{e}}_1$, applying Lemma A.6 for p and η, the probability there is an intersection between $\widetilde{\mathbf{e}}_1$ and $\mathsf{QEnc} \setminus \mathsf{Freq}$ is at most $\frac{1}{2^{\omega(\log(\lambda))}} + \frac{\lambda^{1.1}}{\eta}$.

□

We now prove a lemma analogous to Lemma 4.19.

Lemma C.7. *Suppose $|C| \leq \frac{3\lambda(n-1)}{2}$, where C is the CKE ciphertext. For any constant $c > 0$, assuming $\eta' \geq n\lambda^{c+1}$ and $\eta \geq n\eta'\lambda^{1.1+c}$,* $\Pr[\mathsf{Evnt}_2] \leq \frac{5}{\lambda^c}$.

Proof. The proof follows similarly to that of 4.19, except in the way Lemma C.1 will be invoked. First, similarly to Lemma C.1

$$\Pr[\mathsf{Surprise} \vee \mathsf{Spoof} \vee \mathsf{Intersect}] \leq \frac{4}{\lambda^c}, \tag{9}$$

obtained from the way in which η and η' are instantiated. We will now show whenever all the events $\mathsf{Evnt}_2 \wedge \overline{\mathsf{Surprise}} \wedge \overline{\mathsf{Spoof}} \wedge \overline{\mathsf{Intersect}}$ hold, we can build a forger in the sense of Lemma C.1. The desired bound will then follow.

Let $\mathsf{Chal} := \{(\mathsf{pk}_1, c_1), \ldots, (\mathsf{pk}_m, c_m)\}$ be the set of public key/ciphertexts built up by Brk. Notice that it might be that $m > n$ (because upon hitting Line 4(e)v, Brk continues the execution while pretending the answer is $\perp$, while adding the pair to Chal), and that Chal contains pairs that do not cause a contradiction, when applying Lemma C.1. So, we have to remove some pairs from Chal, as explained below. To apply Lemma C.1, sample randomness R for generating $(\mathsf{CRS}, \mathsf{PK}_1, \ldots, \mathsf{PK}_n)$. Let $\mathsf{A}^{\mathbf{g},\mathbf{e},\mathbf{d}}$ be an adversary that has R hardwired, and which uses R to generate $(\mathsf{CRS}, \mathsf{PK}_1, \ldots, \mathsf{PK}_n)$, and which outputs C formed as $(C, *) \leftarrow_\$ \mathrm{Comm}^{\mathbf{g},\mathbf{e}}(\mathsf{PK}_1, \ldots, \mathsf{PK}_n)$ using fresh randomness. Now let $\mathsf{B}^{\mathbf{g},\mathbf{e},\mathbf{d},\mathbf{u},\mathbf{v}}(C)$, also getting R hardwired in, be an adversary that uses R to re-generate $(\mathsf{CRS}, \mathsf{PK}_1, \ldots, \mathsf{PK}_n)$ and will then simulate $\mathsf{Brk}(\mathsf{PK}_1, \ldots, \mathsf{PK}_n, C)$ to get Chal. Now for every pair (pk, c) in Chal such that there was a query $((\mathsf{pk}, *, *) \underset{\mathbf{e}}{\rightarrow} c) \in \cup_i \mathsf{QGen_i} \cup \mathsf{QC}$ made by B, it will remove (pk, c) from Chal. The fact that Evnt_2 holds implies that Chal remains with at least n pairs, none of which was generated as a result of a previous $\mathbf{e}$ query, by B. From this point on, using exactly the same arguments in Lemma 4.19, we can conclude that all pairs in Chal are distinct and valid. The proof is now complete. □

Now that all the events have been bounded, we can prove an analogous statement to that of Lemma 4.20, hence proving a lowerbound on the probability of attack success. This will imply Lemma 4.4.

D Proof of CGKA Lower Bound

Proof of Theorem 3.1. We build a CKE construction that internally uses a CGKA scheme to execute a CGKA operation sequence. For conducting a CKE commit to k public keys, this operation sequence contains at least one *collective update assistance* for *k passive users*. The core idea of the CKE construction is that precisely the *effective operations* of this *collective update assistance* in the CGKA sequence are embedded in the committed CKE ciphertext. Hence, the total size of these *effective CGKA operations* equals the size of the CKE ciphertext. All remaining operations in the CGKA sequence are, in different shapes, encoded in the CKE common reference string CRS.

As part of the proof, we reduce the security of this CKE construction to the security of the underlying CGKA scheme. Finally, we show that a CGKA scheme that executes this sequence without inducing a communication overhead of $\Omega(k)$ for the *effective operations* implies a CKE construction with compact ciphertexts.

CKE Construction. For clarity, we build a CKE construction that always commits to k public keys, where $k \in \mathsf{poly}(\lambda)$ is fixed and λ is the *security parameter.* This CKE construction can be instantiated with any CGKA operation sequence Seq that adds k passive users and performs at least one subsequent *collective update assistance.*

Internally, the CKE construction executes CGKA operation sequence Seq. All *effective operations* of the included *collective update assistance* in that sequence are embedded in the committed CKE ciphertext. The *pre-add phase* as well as random coins for the remaining operations in that sequence (i.e., *add operations* and *ineffective pre-assistance operations*) are embedded in the CKE common reference string CRS. When processing a received CKE ciphertext to derive the exchanged CKE key, all receivers independently execute the same CGKA operation sequence Seq internally. Firstly, the *pre-add phase* is decoded from the CRS and processed. Then, random coins are decoded from the CRS with which the k *add operations* and the subsequent *ineffective pre-assistance operations* are locally executed and then processed. Finally, the CGKA ciphertexts of the *effective operations* are decoded from the received CKE ciphertext and then processed in order to compute the CGKA key of the last operation in the *collective update assistance.* This CGKA key is output as the derived CKE key.

Let k be the fixed number of input public keys for CKE commits, and Seq be a CGKA execution schedule that adds k *passive users* from the t_1^A th until the t_k^A th operation and afterwards performs a *collective update assistance* until the t^* th operation. Without loss of generality, sequence Seq ends with the t^* th operation such that the *irrelevant end* of that sequence is disregarded. Then PU is the set of passive users' public keys in that sequence, AU_{t^*} is the set of active users' public keys, and EO_{t^*} is the set of effective operations.

- CRSGen:

  1. Execute $(\mathsf{PK}, \mathsf{ST}) \leftarrow_\$ \text{Gen}$ for all users in sequence Seq except for those in set PU
  2. Add $(\mathsf{PK}, \mathsf{ST})$ to CRS for all users in set AU_{t^*}
  3. Add only PK to CRS for all users not in set $PU \cup AU_{t^*}$
  4. Sample random coins for all operations in Seq and execute the sub-sequence that ends with the $t_1^A - 1$ th operation with their coins
  5. Add the random coins for all operations to CRS that were initiated by users in set AU_{t^*} except for the random coins of operations in set EO_{t^*}
  6. Add the output ciphertexts of all operations to CRS that were initiated by users not in set $PU \cup AU_{t^*}$
  7. Add the description of Seq to the CRS
  8. Return CRS

- Init(CRS):

  1. Execute $(\mathsf{PK}, \mathsf{ST}) \leftarrow_\$ \text{Gen}$
  2. Return $(\mathsf{PK}, \mathsf{SK}) = (\mathsf{PK}, \mathsf{ST})$

- Comm($\mathsf{CRS}, \{\mathsf{PK}_i\}_{i \in [k]}$):

  1. Execute sequence Seq until the $t_1^A - 1$ th operation with the secret states of users in set AU_{t^*}, respective random coins, and ciphertexts from CRS.

2. Execute sequence Seq from the t_1^A th to the t_k^A th operation with the secret states of users in set AU_{t^*} and the respective random coins from CRS. Each of the k operations that add users in set PU takes a public key from input $\{\mathsf{PK}_i\}_{i\in[k]}$ as actually added CGKA user in a distinct order
3. Execute sequence Seq from the $t_k^A + 1$ th to the t^* th operation with the secret states of users in set AU_{t^*} from CRS. Operations not in set EO_{t^*} use their respective random coins from CRS and operations in set EO_{t^*} use freshly sampled random coins
4. Use the CGKA key output by the t^* th operation as CKE key K
5. Compose the CKE ciphertext C from the list of ciphertexts output by operations in set EO_{t^*}
6. Return (K, C)

- Derive($\mathsf{CRS}, \mathsf{SK}, \{\mathsf{PK}_i\}_{i\in[k]}, C$):

  1. Execute steps 1 and 2 from algorithm Comm identically
  2. Execute and/or process sequence Seq from the $t_k^A + 1$ th to the t^* th operation with the secret states of users in set AU_{t^*} from CRS and secret state ST from input SK. Operations not in set EO_{t^*} use their respective random coins from CRS. Operations in set EO_{t^*} are processed via CGKA algorithm Proc with ciphertexts from input C and secret state ST from input SK
  3. Use the CGKA key output by the t^* th operation as CKE key K
  4. Return K

When we write "execute sequence", we mean that all operations initiated by active users are indeed (re-)computed with their respective secret state and potentially fixed random coins from CRS. This produces the corresponding output ciphertexts. The ciphertexts of all operations—including the ones directly stored in CRS that are initiated by users who are neither marked active nor passive—are then used for processing the sequence. The sequence is processed with the secret states of the active *and*, in CKE algorithm Derive, passive users. The only operations by active users that are never re-processed with their own states nor re-initiated after being initiated once in CKE algorithm Comm are the effective operations in set EO_{t^*}. The effective operations in set EO_{t^*} are initiated once in CKE algorithm Comm and processed once per CKE receiver in CKE algorithm Derive.

Security of CKE Construction.

Lemma D.1 (CKE Security)**.** *Let* $k \in \mathsf{poly}(\lambda)$ *be fixed and* Seq *be a CGKA operation schedule with* $|PU| = k$ *and with a* collective update assistance *ending with the* t^* *th operation that instantiates CKE construction* CKE*. For every adversary* $\mathcal{A}$ *that is successful according to Definition 2.4 in breaking the security of* CKE *there exists an adversary* $\mathcal{B}$ *that is successful according to Definition 2.3 in breaking the security of the underlying CGKA construction such that* $\mathsf{Adv}(\mathcal{A}) \le \mathsf{Adv}(\mathcal{B})$*.*

Proof of Lemma D.1. We define adversary $\mathcal{B}$ as follows: Given the CGKA execution schedule Seq that instantiates CKE construction CKE, $\mathcal{B}$ composes its query to the CGKA security game by extending this schedule. This extended schedule Seq' additionally specifies that all active users in set AU_{t^*} are corrupted twice: (1) immediately after their public-key secret-state pairs are generated initially and (2) immediately before their respective first effective operation in set EO_{t^*} begins. Furthermore, $\mathcal{B}$ specifies the t^* th operation as the one that establishes the targeted key. The CGKA security game responds on (Seq', t^*) with transcript Trans that contains the following information:

- Public keys of all involved users
- Initial secret states of all active users in set AU_{t^*} (from the first corruption)
- Random coins of all operations initiated by the active users, except for the random coins of effective operations in set EO_{t^*} (from the second corruption)
- Ciphertexts of all operations

Using this information, $\mathcal{B}$ composes CKE common reference string CRS, CKE public keys $\{\mathsf{PK}_i\}_{i\in[k]} = PU$, and CKE ciphertext C. $\mathcal{B}$ invokes $\mathcal{A}(\mathsf{CRS}, \{\mathsf{PK}_i\}_{i\in[k]}, C)$, which returns its guessed key K. This guessed CKE key K is forwarded to the CGKA security game as a guess for the CGKA key that is exchanged with the t^* th operation.

Reduction $\mathcal{B}$ perfectly simulates construction CKE and extracts $\mathcal{A}$'s CKE solution to solve the CGKA challenge. Since every successful adversary $\mathcal{A}$ is reduced to a successful adversary $\mathcal{B}$, we have $\mathsf{Adv}(\mathcal{A}) \leq \mathsf{Adv}(\mathcal{B})$, which proves Lemma D.1. □

Communication Complexity of CGKA. Given fixed $k \in \mathsf{poly}(\lambda)$, a CGKA operation schedule Seq with $|PU| = k$ and a *collective update assistance* ending with the t^* th operation that instantiates CKE construction CKE. The CKE ciphertext in construction CKE precisely contains the CGKA ciphertexts of effective operations EO_{t^*} that realize the corresponding collective update assistance. Assume there exists a CGKA scheme for which the effective operations compute ciphertexts with total size $o(k)$. Using this CGKA scheme, we obtain a CKE construction CKE with compact ciphertexts, which contradicts Theorem 4.5 and, consequently, proves Theorem 3.1. □

E From CKE to CGKA Tightly

In this section, we show to build CGKA from CKE such that the worst-case communication complexity of the CGKA scheme is asymptotically proportional to that of the CKE scheme. Together with the result of Section 3, this shows that the worst-case size of CKE ciphertexts *both* lower bounds *and* upper bounds the worst-case size of CGKA ciphertexts. Note: our CGKA scheme only achieves the security of Definition 2.3; i.e., we do not prove FS properties nor adaptive security for it. However, it is easy to see that it achieves FS similar (slightly weaker) to that of MLS [8] (after an Up operation, a subsequent corruption of the corresponding user does not reveal group keys from before the operation) and that a slightly stronger CKE definition would achieve adaptive security for CGKA.

To build CGKA from CKE we, for simplicity, only consider those CKE schemes without a CRS (i.e., those CKE schemes satisfying Definition 2.4 with $\mathsf{CRS} = \epsilon$). We thus in this section omit CRS from the CKE syntax. Note that of course our black-box lower bound on CKE from PKE in Section 4 still holds for such CKE schemes.

Using such a CKE scheme to build CGKA is straight-forward: All CGKA users during Gen set $(\mathsf{ST}, \mathsf{PK})$ to be the outputs $(\mathsf{SK}, \mathsf{PK})$ of the CKE algorithm Init. A group member that executes a

CGKA operation simply collects the public keys of the users who will be in the group after their operation, inputs them to the CKE Comm algorithm, and outputs the resulting CKE message C and group key K. All other group members can then use CKE algorithm Derive and their CKE secret key to determinstically derive the group key K in Proc.

Formally, each user will store M, the set of public keys of the current group members. Before, a given user is added to the group, they will simply store $\mathsf{M} = \{\mathsf{PK}\}$, where PK is their own public key. $\mathsf{CGKA} = (\text{Gen}, \text{Add}, \text{Rem}, \text{Up}, \text{Proc})$ is defined as follows:

- Gen() executes $(\mathsf{SK}', \mathsf{PK}') \leftarrow_\$ \text{Init}()$ and outputs $(\mathsf{ST}, \mathsf{PK}) = ((\mathsf{SK}', \mathsf{PK}', \mathsf{M}), \mathsf{PK}')$, with $\mathsf{M} = \{\mathsf{PK}'\}$.

- $\text{Add}(\mathsf{ST}, \mathsf{PK}_{\text{add}})$ first parses $\mathsf{ST} = (\mathsf{SK}', \mathsf{PK}', \mathsf{M})$ and sets $\mathsf{M}' \leftarrow \mathsf{M} \cup \{\mathsf{PK}_{\text{add}}\}$, $\mathsf{ST}' \leftarrow (\mathsf{SK}', \mathsf{PK}', \mathsf{M}')$. Then, it executes $(K, C') \leftarrow_\$ \text{Comm}(\mathsf{M}')$, sets $C_G \leftarrow (C', \mathsf{PK}_{\text{add}}, \epsilon), C_B \leftarrow (\mathsf{PK}', \mathsf{PK}_{\text{add}}, \epsilon)$ and outputs $(\mathsf{ST}', K, (C_G, C_B))$.

- $\text{Rem}(\mathsf{ST}, \mathsf{PK}_{\text{rem}})$ first parses $\mathsf{ST} = (\mathsf{SK}', \mathsf{PK}', \mathsf{M})$ and sets $\mathsf{M}' \leftarrow \mathsf{M} \backslash \{\mathsf{PK}_{\text{rem}}\}$, $\mathsf{ST}' \leftarrow (\mathsf{SK}', \mathsf{PK}', \mathsf{M}')$. Then, it executes $(K, C') \leftarrow_\$ \text{Comm}(\mathsf{M}')$, sets $C_G \leftarrow (C', \epsilon, \mathsf{PK}_{\text{rem}}), C_B \leftarrow (\mathsf{PK}', \epsilon, \mathsf{PK}_{\text{rem}})$ and outputs $(\mathsf{ST}', K, (C_G, C_B))$.

- $\text{Up}(\mathsf{ST})$ first parses $\mathsf{ST} = (\mathsf{SK}_{\text{old}}, \mathsf{PK}_{\text{old}}, \mathsf{M})$ and executes $(\mathsf{SK}_{\text{new}}, \mathsf{PK}_{\text{new}}) \leftarrow_\$ \text{Init}()$. It next sets $\mathsf{M}' \leftarrow (\mathsf{M} \setminus \{\mathsf{PK}_{\text{old}}\}) \cup \{\mathsf{PK}_{\text{new}}\}$, $\mathsf{ST}' \leftarrow (\mathsf{SK}_{\text{new}}, \mathsf{PK}_{\text{new}}, \mathsf{M}')$. Then it executes $(K, C') \leftarrow_\$ \text{Comm}(\mathsf{M}')$, sets $C_G \leftarrow (C', \mathsf{PK}_{\text{new}}, \mathsf{PK}_{\text{old}}), C_B \leftarrow (\mathsf{PK}_{\text{old}}, \mathsf{PK}_{\text{new}}, \mathsf{PK}_{\text{old}})$ and outputs $(\mathsf{ST}', K, (C_G, C_B))$.

- $\text{Proc}(\mathsf{ST}, C_G, (\mathbf{B}))$[16] first parses $\mathsf{ST} = (\mathsf{SK}', \mathsf{PK}', \mathsf{M})$ and $C_G = (C', \mathsf{PK}_{\text{add}}, \mathsf{PK}_{\text{rem}})$. Next:

  1. If $\mathsf{M} = \{\mathsf{PK}'\}$, meaning the executing user was added in this operation, she computes the current group member public key set M' from the ciphertexts C_B that have been posted to the bulletin board $\mathbf{B}$ after each operation, then sets $\mathsf{ST}' \leftarrow (\mathsf{SK}', \mathsf{PK}', \mathsf{M}')$.
  2. If $\mathsf{PK}' = \mathsf{PK}_{\text{rem}}$, the algorithm sets $\mathsf{M}' \leftarrow \{\mathsf{PK}'\}$ and $\mathsf{ST}' \leftarrow (\mathsf{SK}', \mathsf{PK}', \mathsf{M}')$, and outputs $(\mathsf{ST}', \perp)$.
  3. Otherwise, the algorithm sets $\mathsf{M}' \leftarrow (\mathsf{M} \setminus \{\mathsf{PK}_{\text{rem}}\}) \cup \{\mathsf{PK}_{\text{add}}\}$ and $\mathsf{ST}' \leftarrow (\mathsf{SK}', \mathsf{PK}', \mathsf{M}')$.[17]

  If Step 2 above was not executed, $K \leftarrow \text{Derive}(\mathsf{SK}', C', \mathsf{M}')$ is executed by the algorithm and (ST', K) is then output.

Theorem E.1. *If* $\mathsf{CKE} = (\text{Init}, \text{Comm}, \text{Derive})$ *is a correct and secure compact key exchange protocol, then* $\mathsf{CGKA} = (\text{Gen}, \text{Add}, \text{Rem}, \text{Up}, \text{Proc})$ *defined above is a correct and secure continuous group key agreement protocol.*

Proof. Correctness of CGKA clearly follows from the correctness of CKE: the CGKA operation executor simply executes Comm on input the current group members' public keys to compute group key K and message C, from which all other group members can compute K using Derive. The executor also includes in the output direct CGKA message C_G: the public key of the added user during an Add operation, the public key of the removed user during a Rem operation, and

[16] Recall that Proc only takes in $\mathbf{B}$ for added users.

[17] We assume $\{\epsilon\} \equiv \emptyset$ when considering set operations including $\{\epsilon\}$.

her old and new public key during an Up operation, so that all group members can keep track of the current pubic keys of other members. This information is additionally included in the bulletin board message C_B, so that added users can easily obtain the current group member public key set from the bulletin board $\mathbf{B}$.

For *non-adaptive* security of CGKA, we will show a reduction from the security of CKE. Assume there exists some adversary $\mathcal{A}$ of CGKA that succeeds in the CGKA security game with probability $\varepsilon > \mathsf{negl}(\lambda)$. We will use $\mathcal{A}$ to construct adversary $\mathcal{B}$ that has advantage ε in the CKE security game, a contradiction.

First note that from operation sequence Seq and challenge epoch t^* which $\mathcal{A}$ provides upon initialization of the CGKA game, $\mathcal{B}$ can easily compute the number of users n in the group at epoch t^*, forward this to its challenger, and assign to those users at epoch t^* public keys $\mathsf{PK}_1, \ldots, \mathsf{PK}_n$ received from the CKE challenger. When $\mathcal{A}$ queries $\mathbf{Gen}()$, if the resulting public key will not be in the group in challenge epoch t^* (which $\mathcal{B}$ can discern from Seq), then $\mathcal{B}$ simply executes $(\mathsf{SK}, \mathsf{PK}) \leftarrow_\$ \mathrm{Init}()$ and sends PK to $\mathcal{A}$. Otherwise, it simply sends the corresponding PK_i from the CKE challenger to $\mathcal{A}$. When $\mathcal{A}$ queries one of $\mathbf{Add}(\mathsf{PK}, \mathsf{PK}^*)$ or $\mathbf{Rem}(\mathsf{PK}, \mathsf{PK}^*)$ for epoch $t \neq t^*$, $\mathcal{B}$ simply executes $C' \leftarrow_\$ \mathrm{Comm}(\mathsf{M}')$, constructs C_G and C_B, then sends (C_G, C_B) to $\mathcal{A}$. For query $\mathbf{Up}(\mathsf{PK})$ for epoch $t \neq t^*$, if the resulting public key will not be in the group in challenge epoch t^* (which $\mathcal{B}$ can discern from Seq), then $\mathcal{B}$ first computes $(\mathsf{SK}_{\mathrm{new}}, \mathsf{PK}_{\mathrm{new}}) \leftarrow_\$ \mathrm{Init}()$, then continues as above. Otherwise, it instead uses the corresponding PK_i from the CKE challenger in place of $\mathsf{PK}_{\mathrm{new}}$, then continues as above. For the epoch t^* query, $\mathcal{B}$ simply uses the challenge ciphertext C from the CKE challenger, constructs C_G and C_B, then sends (C_G, C_B) to $\mathcal{A}$. For $\mathbf{Corr}(\mathsf{PK})$ queries from $\mathcal{A}$, $\mathcal{B}$ simply returns the corresponding SK, and the random coins which it sampled for the corresponding user's executions of Comm() and Init(). When $\mathcal{A}$ sends K, $\mathcal{B}$ simply forwards it to its challenger.

Clearly, $\mathcal{B}$ has perfectly simulated the CGKA security game against CGKA for $\mathcal{A}$. Namely, since the epoch t^* which $\mathcal{A}$ challenges must not be in **WeakEpochs**, it must be that all users in the group at epoch t^* were either never corrupted, or were updated since their last corruption. Thus, all corruptions queried by $\mathcal{A}$ correspond to secret states and randomness which $\mathcal{B}$ generated and sampled, respectively, on its own. Thus since $\mathcal{A}$ wins the CGKA game with probability ε, $\mathcal{B}$ wins the CKE game against CKE with probability ε, a contradiction. □

The following corollary follows immediately from the construction of CGKA above:

Corollary E.2. *The asymptotic worst-case communication complexity of* CGKA *operations with* n *group members is equivalent to the asymptotic worst-case communication complexity of* CKE Comm() *algorithm on input* n *public keys.*

F Tainted TreeKEM Summary

Here we give a summary of Tainted TreeKEM (TTKEM). We take it almost verbatim from Alwen et al. [6, §2], adapting it to our notation, and removing details which are not relevant to our paper.

F.1 Overview

TTKEM works over a binary tree T and makes black-box use of PKE protocol PKE $=$ (Gen, Enc, Dec) and PRF F. The nodes in the tree are associated with the following values:

- a seed Δ;
- (all nodes except the root) a PKE secret/public key pair $(\mathsf{sk}, \mathsf{pk})$ derived deterministically from the seed; and
- (all nodes except leaves and root) a tainter ID.

The root has no associated public/secret key pair, instead its seed is the current group key.

To achieve FS and PCS, and to manage group membership, it is necessary to constantly renew the secret keys used in the protocol. We will do this through the group operations Up(), Rem(), and Add(). We will use the term *refresh* to refer to the renewal of a particular (set of) key(s) (as opposed to the group operation). Each group operation will refresh a part of the tree, always including the root and thus resulting in a new group key which can be decrypted by all members of the current group.

Due to our simplified CGKA definition, each group member has a consistent view of the public information in the tree, namely public keys, tainter IDs and past operations. Furthermore, group members will have a partial view of the secret keys. More precisely, every user has an associated *protocol state* ST (or state for short when there is no ambiguity), which represents everything users need to know to stay part of the group. In particular, we define a state as the double $\mathsf{ST} = (\mathsf{M}, \mathsf{T})$, where

- M denotes the set of group members, i.e. PK's that are part of the group; and
- T denotes a binary tree defined as above, where for each group member, their PK is associated to a leaf node.

As mentioned, a user will generally not have knowledge of the secret keys associated to all tree nodes. However, if they add or remove parties, they will potentially gain knowledge of secret keys outside their path. We observe that this will not be a problem as long as we have a mechanism to keep track of those nodes and refresh them when necessary, towards this end we introduce the concept of tainting.

Tainting. Whenever party i refreshes a node not lying on their path to the root, that node becomes *tainted* by party i. Whenever a node is tainted by a party i, that party has potentially had knowledge of its current secret in the past. So, if party i was corrupted in the past, the secrecy of that value is considered compromised (even if she deleted that value right away and is no longer compromised). Even worse, all values that were encrypted to that node are compromised too. We will assign a tainter ID to all nodes. This can be empty, i.e. the node is untainted, or corresponds to a single party i, who last generated this node's secret but is not supposed to know it. The tainter ID of a node is determined by the following simple rules.

- After party i initialises, all internal nodes not on her path become tainted by her.[18]
- If party i updates or removes someone, refreshed nodes on her path become untainted.
- If party i updates or removes someone, all refreshed nodes not on her path become tainted by her.

[18] Our CGKA syntax only allows party i to one-by-one add other users to the group, instead of an explicit initialisation algorithm, but we keep this here for completeness.

Hierarchical derivation of updates. When refreshing a whole path we sample a seed Δ_0 and derive all the secrets for that path from it. This way, we reduce the number of decryptions needed to process the update, as parties only need to recover the seed for the "lowest" node that concerns them, and then can derive the rest locally. To derive the different new secrets we follow the specification of TreeKEMv9 [8]. Essentially, we consider a PRF F, fix tag x, and together with Gen, we derive the keys for the nodes as follows:

$$(\mathsf{sk}_i, \Delta_{i+1}) \coloneqq F(\Delta_i, x)$$

$$\mathsf{pk}_i \leftarrow \mathrm{Gen}(\mathsf{sk}_i)$$

where Δ_i is the seed for the ith node (the leaf being the 0th node, its parent the 1st etc.) on the path and $(\mathsf{sk}_i, \mathsf{pk}_i)$ its new key pair.

With the introduction of tainting, it is no longer the case that all nodes to be refreshed lie on a path. Hence, we partition the set of all the nodes to be refreshed into paths and use a different seed for each path. For this we need a unique path cover, as users processing the update will need to know which nodes secrets depend on which. A concrete example is given in [6, §A.2], but any unambiguous partition suffices. The only condition required is that the updating of paths is done in a particular common order that allows for encryptions to to-be-refreshed nodes to be done under the respective updated public key (one cannot hope for PCS otherwise).

Let us stress that a party processing an update involving tainted nodes might need to retrieve and decrypt more than one encrypted seeds, as the refreshed nodes on its path might not all be derived hierarchically. Nonetheless, party needs to decrypt at most $\log n$ ciphertexts in the worst case.

F.2 TTKEM Dynamics

Whenever a user i wants to perform a group operation, she will generate and send the appropriate Update, Add or Remove message to all group members, and post the appropriate information to the bulletin board. Messages should contain the identity of the sender, the operation type, encryptions of the new seeds, and any new public keys. A more detailed description, as well as pseudo-code for the distinct operations is presented in [6, §A.3].

Initialize. To create a new group with user public keys $\mathsf{M} = \{\mathsf{PK}_1, \ldots \mathsf{PK}_n\}$, user 1 generates a new tree T, where the leaves have the associated public keys corresponding to the group members.[19] The group creator then samples new key pairs for all the other nodes in T (optimizing with hierarchical derivation) and crafts welcome messages for each party. These welcome messages should include an encryption of the seed that allows the computation of the keys of the appropriate path. Each party will download from the bulletin board tree T.

Add. To add a new user with PK_j to the group, user i identifies a free spot for them, samples a seed Δ, and derives seeds for the nodes along the path to the root. She then encrypts the new seeds to all the nodes in the co-path (one ciphertext per node suffices given the hierarchical derivation) and sends them over together with the public key PK_j of the added party.

[19] Again, our CGKA syntax does not have an explicit initialisation algorithm, but we keep this here for completeness.

Update. To perform an Update, a user computes a path partition for the set of nodes not on her path that need to be refreshed (nodes tainted or with a tainted ancestor), samples a seed per such path, plus a seed for their path, and derives the new key-pairs for each node, as described above. She then encrypts the secret keys under the appropriate public keys in the copaths.

Remove. To remove user with PK_j, user i performs an Update as if it was user j, refreshing all nodes in user j's path to the root, as well as all her tainted nodes (which will become tainted by user i after the removal).

Process. When a user receives a protocol message C, it identifies which kind of message it is and performs the appropriate update of their state, by updating the list of participants if necessary, overwriting any keys, and updating the tainted ID's.