


On the Worst-Case Inefficiency of CGKA

Alexander Bienstock¹, Yevgeniy Dodis^{1*}, Sanjam Garg^{2**}, Garrison Grogan³,
Mohammad Hajiabadi^{4***}, and Paul Rösler¹

¹ New York University, {abienstock,dodis,paul.roesler}@cs.nyu.edu

² UC Berkeley and NTT Research, sanjamg@berkeley.edu

³ garrisonwgrogan@gmail.com

⁴ University of Waterloo, mdhajiabadi@uwaterloo.ca

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 members. 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.

The full version [10] of this article is available in the IACR eprint archive as article [2022/1237](#).

* Partially supported by gifts from VMware Labs and Algorand Foundation, and NSF grants 1815546 and 2055578.

** 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.

*** Work supported by an NSERC Discovery Grant RGPIN/03270-2022.

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 [21] 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 [32,26,2,20,28,14,18]. 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 [30]. 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 [3,6,21,1] 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*. 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 important¹ and ultimately prioritized over efficiency² 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 [3,6,21,1], 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 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 .³ If user B removes user A as the next operation, then due

¹ First proposal of the TreeKEM design with a discussion about the double-join problem: <https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8/>

² Proposal to prevent double-joins in TreeKEM, resulting in linear complexity in the worst-case: <https://mailarchive.ietf.org/arch/msg/mls/Zzw2tqZC1FCbVZA9LKERSMIQXik/>

³ 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).

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 recovery⁴) is entirely independent with respect to our employed methods and resulting statement.

CKE tightly implies CGKA. For completeness, in the full version [10] 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, CGKA and CKE are surprisingly equivalent in terms of both cryptographic strength *and* worst-case (communication) complexity; if one could construct CKE efficiently, they could also construct CGKA efficiently, and vice versa.

⁴ Unlike in [1] who circumvent the [11] lower bound by allowing for slower PCS recovery.

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 our 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 7) 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 (\mathbf{pk}, m, r)). This in particular implies that in order to get a valid (\mathbf{pk}, c) —one under which there exists some m and r such that $\mathbf{e}(\mathbf{pk}, m, r) = c$ —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, \dots, c_t with corresponding public keys $\mathbf{pk}_1, \dots, \mathbf{pk}_t$, 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 $(\mathbf{pk}_1, \dots, \mathbf{pk}_t)$. 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 $\mathbf{PK}_1, \dots, \mathbf{PK}_n$ (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 [22, 19] 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 $(\text{CRSGen}^{\mathbf{g}}, \text{Init}^{\mathbf{g}}, \text{Comm}^{\mathbf{e}}, \text{Derive}^{\mathbf{d}})$. 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 the full version [10].

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 [31,33,24,17,25,29,13]. 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?⁵ 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.

⁵ If neither administrator is removed, of course $O(\log n)$ communication can be retained if they share a multicast tree.

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 $y \leftarrow_{\S} A(x)$. We will also use the notation $x \leftarrow_{\S} X$ 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 .

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 1. A Continuous Group Key Agreement *scheme* $\text{CGKA} = (\text{Gen}, \text{Add}, \text{Rem}, \text{Up}, \text{Proc})$ consists of the following algorithms:⁶

- *Gen* is a PPT algorithm that outputs (ST, PK) .
- *Add* is a PPT algorithm that takes in (ST, 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 (ST, 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 .

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 **Gen**(), **Add**(), **Rem**(), **Up**(), and **Corr**() that will be executed. **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 **Add**() to initialize the group. It keeps track of group members for each epoch using dictionary \mathbf{G} . For simplicity, in each group

⁶ For the sake of comprehensible communication analysis, we do not provide an explicit $\text{Create}(\text{ST}, \text{PK}_1, \dots, \text{PK}_n)$ algorithm (for which in practice, $\Omega(n)$ ciphertext size could be tolerated). Instead, we require the group creator to one-by-one add $\text{PK}_1, \dots, \text{PK}_n$, 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) $\mathbf{G}[*], \mathbf{Rand}[*], \mathbf{ST}[*], \mathbf{K}[*] \leftarrow \perp$.

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

Fig. 1. The CGKA correctness and security games.

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. 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 $\text{PK}, \text{PK}' \in \mathbf{G}[t]$: $\Pr [\mathbf{K}[t, \text{PK}] = \mathbf{K}[t, \text{PK}']] = 1$.

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.⁷ 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., $(\mathbf{WeakUsers} \cap \mathbf{G}[t+1]) \neq \emptyset$), 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 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:

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

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 .

⁷ Analyzing the effect of required authenticity under weak randomness [7] on (communication) complexity in the group setting [27], as well as of extended security goals such as anonymity [23] remains an interesting open question.

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 $1, \dots, n$ to sample independent (across users) key pairs $(SK_1, PK_1), \dots, (SK_n, PK_n)$, then publicly broadcast PK_1, \dots, 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 $\{PK_j\}_{j \in [n]}$, and C to derive K .

Definition 4. A Compact Key Exchange scheme $\text{CKE} = (\text{CRSGen}, \text{Init}, \text{Comm}, \text{Derive})$ in the standard model with common reference string $\text{CRS} \in \text{CRS}$ consists of the following algorithms:

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

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

$$\begin{aligned} \Pr [K \leftarrow \text{Derive}(\text{CRS}, SK_i, \{PK_j\}_{j \in [n]}, C) : (K, C) \leftarrow_{\$} \text{Comm}(\text{CRS}, \{PK_j\}_{j \in [n]}) ; \\ \forall j \in [n], (SK_j, PK_j) \leftarrow_{\$} \text{Init}(\text{CRS}); \\ \text{CRS} \leftarrow_{\$} \text{CRSGen}(1^\lambda)] = 1. \end{aligned}$$

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

$$\begin{aligned} \Pr [K \leftarrow_{\$} \mathcal{A}(\text{CRS}, \{PK_i\}_{i \in [n]}, C) : (K, C) \leftarrow_{\$} \text{Comm}(\text{CRS}, \{PK_i\}_{i \in [n]}) ; \\ \forall i \in [n], (SK_i, PK_i) \leftarrow_{\$} \text{Init}(\text{CRS}); \\ \text{CRS} \leftarrow_{\$} \text{CRSGen}(1^\lambda)] \leq \text{negl}(\lambda). \end{aligned}$$

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

Remark 1. Of course, there is a simple CKE protocol (without CRS) from PKE scheme $\text{PKE} = (\text{Gen}, \text{Enc}, \text{Dec})$, where $|C| = O(\lambda \cdot n)$: $\text{Init}()$ simply samples $sk \leftarrow_{\$} \{0, 1\}^\lambda$, then computes $pk \leftarrow \text{Gen}(sk)$ and outputs (sk, pk) . $\text{Comm}(\{pk_i\}_{i \in [n]})$ samples $K \leftarrow_{\$} \{0, 1\}^\lambda$, and for each $i \in [n]$ computes $c_i \leftarrow_{\$} \text{Enc}(pk_i, K)$. It then outputs (K, C) , where $C = (c_1, \dots, c_n)$. Finally, $\text{Derive}(sk_i, \{pk_j\}_{j \in [n]}, C)$ computes $K \leftarrow \text{Dec}(sk_i, c_i)$ 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. Due to space constraints, the simpler counter direction—building CGKA from CKE, tightly—is provided in the full version [10]. 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)

Let sequence $\text{Seq} = (\text{Op}_1, \dots, \text{Op}_n)$ 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: $\text{Op}_t = (\text{Up}, \text{PK}, \perp)$ means that PK updates their state; $\text{Op}_t = (\text{Add}, \text{PK}, \text{PK}^*)$ means that PK adds PK^* ; $\text{Op}_t = (\text{Rem}, \text{PK}, \text{PK}^*)$ 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 $\text{PK}^* \in PU$ there exists an operation $(\text{Add}, \cdot, \text{PK}^*)$ but neither an operation $(\text{Rem}, \cdot, \text{PK}^*)$ nor an operation $(\cdot, \text{PK}^*, \cdot)$ 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 $\text{Op}_{t_1^A} = (\text{Add}, \cdot, \text{PK}^*)$ is the *first* operation that adds a user $\text{PK}^* \in PU$ to the group. (2.a) The *add operations*, starting with operation $\text{Op}_{t_1^A}$, end with the *last* operation $\text{Op}_{t_k^A} = (\text{Add}, \cdot, \text{PK}^*)$ that adds a user $\text{PK}^* \in PU$ 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 $\text{PK}^* \in AU_{t^*}$ iff there exists at least one operation $\text{Op}_t = (\cdot, \text{PK}^*, \cdot)$ but no operation $\text{Op}_t = (\text{Rem}, \cdot, \text{PK}^*)$ 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 $\text{Op}_{t^*}, t^* > t_k^A$, for which it must hold that all users $\text{PK}^* \in AU_{t^*}$ 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 $PK \in AU_{t^*}$, the set of *effective operations* EO_{t^*} in a *collective update assistance* contains all operations $Op_{t'} = (\cdot, PK, \cdot)$ that were initiated since the most recent update operation $Op_{t_{PK}} = (Up, PK, \cdot)$ by user PK , where $t_{PK} \leq t' \leq t^*$ 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 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.*

The proof of Theorem 1 is provided in the full version [10].

In Corollary 1 we formulate a simpler, more specific variant of bad sequences that is directly implied by Theorem 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 1 (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 $\Omega(k)$.⁸ As a result, the entire schedule induces a communication overhead of $\Omega(k \cdot l)$. We formally define this CGKA execution schedule in Definition 5 and give the corresponding Corollary 2.

Definition 5 (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*

⁸ We strike out “collective” because each update assistance is conducted by a single active user in this execution schedule.

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 $(\text{Op}_{i,A}, \text{Op}_{i,R}, \text{Op}_{i,U}), i \in [l]$, where $\text{Op}_{i,A} = (\text{Add}, \text{PK}_i, \text{PK}_{i+1})$, $\text{Op}_{i,R} = (\text{Rem}, \text{PK}_{i+1}, \text{PK}_i)$, and $\text{Op}_{i,U} = (\text{Up}, \text{PK}_{i+1}, \perp)$ for independent users $\text{PK}_j, j \in [l+1]$.

Corollary 2 (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 2 is a direct application of Theorem 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. Due to space limitations, many other preliminaries are deferred to the full version [10].

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

Definition 6. *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.*

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

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

Definition 7. A Compact Key Exchange scheme in the Ψ -model is defined equivalently as in Definition 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., $\text{Init}^\Psi(\text{CRS})$. 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 ($\text{CRSGen}, \text{Init}, \text{Comm}, \text{Derive}$), 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 1 (\mathbf{O} is secure against $(\mathbf{O}, \mathbf{u}, \mathbf{v})$). For any polynomial-query adversary \mathbf{A} :

$$\Pr[\mathbf{A}^{\mathbf{O}, \mathbf{u}, \mathbf{v}}(\text{pk}, c) = b] \leq 1/2 + \frac{1}{2^{\lambda/2}}, \text{ where } (\mathbf{g}, \mathbf{e}, \mathbf{d}, \mathbf{u}, \mathbf{v}) \leftarrow_{\$} \Psi, \mathbf{O} := (\mathbf{g}, \mathbf{e}, \mathbf{d}), b \leftarrow_{\$} \{0, 1\}, \text{sk} \leftarrow_{\$} \{0, 1\}^\lambda, \text{pk} = \mathbf{g}(\text{sk}), r \leftarrow_{\$} \{0, 1\}^\lambda \text{ and } c = \mathbf{e}(\text{pk}, b; r).$$

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

Roadmap. Lemma 1 is proved in a straightforward way (hence omitted), given the random nature of the oracles. The proof of Lemma 2 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 $(\text{Init}^{\mathbf{g}}, \text{Comm}^{\mathbf{e}}, \text{Derive}^{\mathbf{d}})$ in Section 4.2, and will then describe an attack against general constructions in the full version [10]. Lemma 2 will follow similarly from the below simpler attack. We may now obtain the following from Lemmas 1, 2, proved via standard black-box separation techniques.

Theorem 2. 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 $(\text{CRSGen}^g, \text{Init}^g, \text{Comm}^e, \text{Derive}^d)$

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

Attack overview. Let $(\text{PK}_1, \dots, \text{PK}_n, C)$ 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 6). 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 describe.

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 $(\text{PK}_i, \text{SK}_i) \leftarrow_{\$} \text{Init}^g(1^\lambda)$ a set of g -type query/answer pairs made. Let $\text{KPair}_i = \{(\text{pk}_{i,1}, \text{sk}_{i,1}), \dots, (\text{pk}_{i,t}, \text{sk}_{i,t})\}$ be the set of public/secret key pairs produced during the generation of PK_i . These public keys are in some way encoded in PK_i , and the ability to decrypt with respect to these base $\text{pk}_{i,j}$ public keys is the only advantage that the i th party, who has SK_i , has over an adversary.

Consider a random execution of $(K, C) \leftarrow_{\$} \text{Comm}^e(\text{PK}_1, \dots, \text{PK}_n)$, and let $Q = \{(\text{pk}_1, b_i, r_i, c_i) \mid i \in [f]\}$ contain the set of all query/answer pairs, and let $Q_c = \{c_1, \dots, c_f\}$. 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 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 $e(\text{pk}, \cdot, \cdot)$. 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 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 : $S'_h \subseteq S'_1 \cup \dots \cup S'_{h-1}$. In other words, the set of base trapdoors needed to decrypt S'_h is a subset of those for $S'_1 \cup \dots \cup S'_{h-1}$. Moreover, in order for any user to be able to decrypt some (pk, c) , the user should have observed a query/answer pair (pk, sk) during its execution of $\text{Init}^g(1^\lambda)$. Thus, recalling KPair_h , the set of base secret keys needed to decrypt elements in S'_h is a subset of $\text{KPair}_1 \cup \dots \cup \text{KPair}_{h-1}$. But each of these KPair_i sets (for $i \in [n]$) is obtained by running $\text{Init}^g(1^\lambda)$ on a security parameter, and so if an adversary runs $\text{Init}^g(1^\lambda)$ 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 (pk, 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 $\text{Derive}(\text{SK}, \cdot)$. 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 $(\text{CRSGen}^{\mathbf{g}}, \text{Init}^{\mathbf{g}}, \text{Comm}^{\mathbf{e}}, \text{Derive}^{\mathbf{d}})$ that we want to prove an impossibility for. The assumption is made only for ease of exposition.

Assumption 3. *We assume for any oracle $(\mathbf{g}, \mathbf{e}, \mathbf{d}) \leftarrow_{\S} \Psi$ picked as in Definition 6, each algorithm in $(\text{CRSGen}^{\mathbf{g}}, \text{Init}^{\mathbf{g}}, \text{Comm}^{\mathbf{e}}, \text{Derive}^{\mathbf{d}})$ makes only a security parameter λ number of queries.*

Definition 8 (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 $O_2 \in \text{Supp}(\Psi)$: $O_1 \subseteq O_2$, 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 \mathbf{S} if $O_1 \cup \mathbf{S}$ 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 9 (Forbidding queries). *Let Forbid consists of “wildcard” queries/responses, of the form $(q \xrightarrow{z} *)$ or $(* \xrightarrow{z} u)$, 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 $(q \xrightarrow{z} *) \in \text{Forbid}$ the oracle \tilde{z} is not defined on input q and (b) for any $(* \xrightarrow{z} u) \in \text{Forbid}$ 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 $\text{Init}^{\mathbf{g}}(\text{CRS})$ 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 3 any execution of $\text{Init}^{\mathbf{g}}(\text{CRS})$ makes exactly λ queries. We say a partial oracle \mathbf{O}' (defined only on a subset of points) is minimal for an execution $\text{Init}^{\mathbf{O}'}(\text{CRS}; R)$, 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 10 (Sampling partial oracles). *We define the procedure ConsOrc . In this definition we assume that the algorithm $\text{Init}^{\mathbf{g}, \mathbf{e}}$ 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:** $(\text{CRS}, \text{PK}, \text{Freq}, \text{Forbid})$: 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 $(q \xrightarrow{z} *)$ or $(* \xrightarrow{z} u)$, where $z \in \{\mathbf{g}, \mathbf{e}\}$.
- **Output:** (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 3), 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 9) (c) $\text{Init}^{\mathbf{O}'}(\text{CRS}; R) = (\text{PK}, \text{SK})$ and (d) \mathbf{O}' is R -minimal: the execution of $\text{Init}^{\mathbf{O}'}(\text{CRS}; R)$ makes only queries to those in \mathbf{O}' , and nothing else. If no such (SK, \mathbf{O}') exists, output \perp .⁹

In our attack, the adversary will try performing simulated decryptions for different parties. The adversary will do so by sampling a simulated secret key $\widetilde{\text{SK}}$ for that party, along with a partial oracle \mathbf{g}' relative to which $\widetilde{\text{SK}}$ is a secret key for that party’s public key PK (i.e., $(\text{PK}, \widetilde{\text{SK}}) \leftarrow_{\S} \text{Init}^{\mathbf{g}'}(\text{CRS})$). 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 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 $(\text{sk}_1 \xrightarrow{\mathbf{g}} \text{pk}_1)$ 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 $\text{Query}(L)$ to denote the query parts of the elements of L .

Definition 11 (Composed Oracles \diamond^*). Let $\mathbf{O} := (\mathbf{g}, \mathbf{e}, \mathbf{d})$ be a Ψ -valid oracle (a possible output of Ψ) and let

$$\mathbf{g}' := \{(\text{sk}_1 \xrightarrow{\mathbf{g}} \text{pk}_1), \dots, (\text{sk}_w \xrightarrow{\mathbf{g}} \text{pk}_w)\}$$

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

⁹ This can happen because of the presence of forbidding queries in Forbid .

Lemma 3. *Assuming \mathbf{O} and \mathbf{g}' are Ψ -valid, $(\tilde{\mathbf{g}}, \mathbf{e}, \tilde{\mathbf{d}})$, obtained as in Definition 11, is Ψ -valid, and hence PKE-valid.*

Due to space limitations, we formally define and analyze the attack in the full version [10].

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 Π' such that on sequences drawn from this distribution, Π' has much lower expected amortized communication cost than Π . 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 1 of Theorem 1, if starting with a group of n users with public keys $\text{PK}_1, \dots, \text{PK}_n$ 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 $1, \dots, \alpha$ with public keys $\text{PK}_1, \dots, \text{PK}_\alpha$) 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 $2, \dots, \alpha$ 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 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 Π is then left with a choice regarding the updates of users $2, \dots, \alpha$. Roughly, either:

- (a) Each of the users $2, \dots, \alpha$ 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 Π chooses strategy (a) or (b) (or probabilistically favors one over the other), it can be starkly outperformed by another protocol Π' when executing certain sequences (by the same amount in both cases). In the case of (a), if after users $2, \dots, \alpha$ execute their updates, the k added users

come online and execute their own updates, then users $2, \dots, \alpha$ 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 Π 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 **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 1, 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 **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, **LazyBad** and **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 **LazyBad** and **ActiveBad** are parameterized by:

- n : The number of users in the group before user 1 adds the new users;
- **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 **ActiveBad** or **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 **LazyBad**, the number of rounds of updates in which one of the originally active users is passive. We use ℓ in **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 **ActiveBad**(n , **PreAddSeq**, k, α, ℓ) is just one sequence (i.e., that sequence is drawn from the distribution **ActiveBad**(n , **PreAddSeq**, k, α, ℓ) 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 **ActiveBad**(n , **PreAddSeq**, k, α, ℓ) as **ActiveBad**(**PreAddSeq**) and **LazyBad**(n , **PreAddSeq**, k, α, ℓ) as **LazyBad**(**PreAddSeq**). We first define **LazyBad**(**PreAddSeq**):

Definition 12. A sequence **Seq** of CGKA operations drawn from distribution **LazyBad**(n , **PreAddSeq**, k, α, ℓ) consists of the following phases:

- **Phase P0:** The pre-add phase, i.e., the operations $\text{Op}_1, \dots, \text{Op}_{t_1^A-1}$ of **PreAddSeq**.
- **Phase P1:** For $i \in [k]$ operations $\text{Op}_{1,i} = (\text{Add}, \text{PK}_1, \text{PK}_{n+i})$. Then operation $\text{Op}_{1,k+1} = (\text{Up}, \text{PK}_1, \perp)$.
- **Phase P2:** For $i \in [\alpha - 1]$ operations $\text{Op}_{2,i} = (\text{Up}, \text{PK}_{i+1}, \perp)$.
- **Phase P3:** Let $j \leftarrow_{\$} [\alpha]$. Then, for each $m \in [\ell]$: for every $i < j$ (resp. $i > j$), $\text{Op}_{3,(m-1)(\alpha-1)+i} = (\text{Up}, \text{PK}_i, \perp)$ (resp. $\text{Op}_{3,(m-1)(\alpha-1)+i-1} = (\text{Up}, \text{PK}_i, \perp)$), where PK_i is the most recent public key of user i .

Next, we define **ActiveBad**(**PreAddSeq**), which has the same phases 0 – 2 as **LazyBad**(**PreAddSeq**), but differs in phase 3 as described above:

Definition 13. A sequence **Seq** of CGKA operations drawn from distribution **ActiveBad**(n , **PreAddSeq**, k, α, ℓ) consists of the same phases P0-P2 as above then:

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

Note that by Theorem 1, for every CGKA protocol it must be that update $\text{Op}_{1,k+1} = (\text{Up}, \text{PK}_1, \perp)$ in Phase P1 of either distribution requires $\Omega(k)$ communication, no matter what the operations of **PreAddSeq** were and what structure the adds of user 1 in Phase 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 $\text{PK}_{n+1} \dots, \text{PK}_{n+\alpha}$ 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 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 P2 and P3:

Definition 14. Let Seq be a sequence of CGKA operations drawn from distribution

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

$$\text{CC}_\Pi(\text{LazyBad}(\text{PreAddSeq})) := \mathbb{E}_{\text{Seq} \leftarrow \text{LazyBad}(\text{PreAddSeq})} [\text{CC}_\Pi[\text{Seq}]]$$

$$(\text{resp. } \text{CC}_\Pi(\text{ActiveBad}(\text{PreAddSeq})) := \mathbb{E}_{\text{Seq} \leftarrow \text{ActiveBad}(\text{PreAddSeq})} [\text{CC}_\Pi[\text{Seq}]]),$$

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 $\text{LazyBad}(\text{PreAddSeq})$ or $\text{ActiveBad}(\text{PreAddSeq})$. First, we define a specific PreAddSeq which intuitively leaves the CGKA group in a *full* state:

Definition 15. Valid sequence of CGKA operations Full_n contains the following operations in order: $(\text{Add}, \text{PK}_1, \text{PK}_2), (\text{Add}, \text{PK}_1, \text{PK}_3), \dots, (\text{Add}, \text{PK}_1, \text{PK}_n), (\text{Up}, \text{PK}_1, \perp), (\text{Up}, \text{PK}_2, \perp), \dots, (\text{Up}, \text{PK}_n, \perp)$.

Theorem 4. Let $\ell = O(k/\log n)$. Then for every CGKA protocol Π and every PreAddSeq , there exists some other protocol Π' such that either

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

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

Note that 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 Π on any PreAddSeq , with upper bounds for protocols Π' on specifically Full_n .

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

Definition 16. *CGKA protocol Π is Lazy if $\Pr[\exists i \in [\alpha - 1] : \mathbf{CC}_\Pi[\text{Op}_{2,i}] = o(k)] > 1/2$. Otherwise, Π is Active.*

Proof of Theorem 4. The following lemmas, proved in the full version [10] due to space limitations, and which intuitively follow from the descriptions of this section, allow us to prove Theorem 4. \square

Lemma 4. *There is a protocol Π_{Active} that has expected amortized communication cost $\mathbf{CC}_{\Pi_{\text{Active}}}(\text{LazyBad}(\text{Full}_n)) = O(k/\ell + \log n)$ on random Seq drawn from $\text{LazyBad}(n, \text{Full}_n, k, \alpha, \ell)$.*

Lemma 5. *For every protocol Π that is Lazy and every PreAddSeq, the expected total communication cost $\mathbf{CC}_\Pi(\text{LazyBad}(\text{PreAddSeq})) = \Omega(k/\alpha^2)$ on random Seq drawn from $\text{LazyBad}(n, \text{PreAddSeq}, k, \alpha, \ell)$.*

Lemma 6. *There is a protocol Π_{Lazy} that has expected total communication cost $\mathbf{CC}_{\Pi_{\text{Lazy}}}(\text{ActiveBad}(\text{Full}_n)) = O(\log n)$ on random Seq drawn from $\text{ActiveBad}(n, \text{Full}_n, k, \alpha, \ell)$.*

Lemma 7. *For every protocol Π that is Active and every PreAddSeq, its expected total communication cost $\mathbf{CC}_\Pi(\text{ActiveBad}(\text{PreAddSeq})) = \Omega(k/\ell)$ on random Seq drawn from $\text{ActiveBad}(n, \text{PreAddSeq}, k, \alpha, \ell)$.*

The following corollary thus easily follows:

Corollary 3. *Let $k = \Omega(n)$, $\ell = \Theta(\sqrt{n})$, and $\alpha = O(\sqrt{\log n})$. Then for every protocol Π , there exists some other protocol Π' such that either on a random sequence drawn from $\text{ActiveBad}(\text{Full}_n)$, or from $\text{LazyBad}(\text{Full}_n)$, Π' has a factor of $\Omega(\sqrt{n}/\log n)$ better amortized communication in expectation than Π 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](https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8), 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. Cryptology ePrint Archive (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. 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)
20. 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)

21. 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)
22. 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)
23. Dowling, B., Hauck, E., Riepel, D., Rösler, P.: Strongly anonymous ratcheted key exchange. In: ASIACRYPT 2022. LNCS (2022)
24. Harney, H., Muckenhirn, C.: Rfc2093: Group key management protocol (gkmp) specification (1997)
25. Mitra, 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>
26. Perrin, T., Marlinspike, M.: The double ratchet algorithm (2016), <https://signal.org/docs/specifications/doubleratchet/>
27. 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)
28. 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)
29. 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)
30. Smart, N.P.: Efficient key encapsulation to multiple parties. In: Blundo, C., Ciamato, S. (eds.) SCN 04. LNCS, vol. 3352, pp. 208–219. Springer, Heidelberg (Sep 2005)
31. Wallner, D., Harder, E., Agee, R.: Rfc2627: Key management for multicast: Issues and architectures (1999)
32. 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)
33. 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>