Keep the Dirt: Tainted TreeKEM, an Efficient and Provably Secure Continuous Group Key Agreement Protocol

Joel Alwen[1], Margarita Capretto[2], Miguel Cueto[3], Chethan Kamath[4⋆], Karen Klein[4⋆], Guillermo Pascual-Perez[4] ⋆⋆, Krzysztof Pietrzak[4⋆], and Michael Walter[4⋆]

[1] Wickr Inc.
[2] Universidad Nacional de Rosario
[3] Universidad de Oviedo
[4] IST Austria

Abstract. While end-to-end encryption protocols with strong security are known and widely used in practice, designing a protocol that scales efficiently to large groups and enjoys similar security guarantees remains an open problem. The only known approaches to date are ART (Cohn-Gordon et al., CCS18) and TreeKEM (IETF, The Messaging Layer Security Protocol, draft). ART enjoys a security proof, albeit with a superexponential bound, and is not dynamic enough for practical purposes. TreeKEM has not been proven secure at this point and can suffer some efficiency issues due to dynamic group operations (i.e. adding and removing users).

As a first contribution we present a variant of TreeKEM, that we call Tainted TreeKEM, which can be more efficient than TreeKEM depending on the distribution of add and remove operations.

Our second contribution is a security proof for Tainted TreeKEM (and also TreeKEM) with a meaningful security bound against active and adaptive adversaries, showing that the protocol supports post compromise security and forward security. Concretely, we achieve an only slightly superpolynomial security loss of $q^{\log\log(n)}$, where n is the group size and q the total number of (update/remove/invite) operations.

⋆ Funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (682815 - TOCNeT)

⋆⋆ Funded by European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 665385.

Table of Contents

1 Introduction

Messaging systems allow parties to communicate asynchronously, so the parties need not be online at the same time. The exchanged messages are buffered by an untrusted delivery server, and then relayed to the receiving party when it comes online. Secure messaging protocols (like Open Whisper Systems' Signal Protocol) provide not only end-to-end privacy and authenticity, but by having the parties perform regular key updates, they even achive stronger security guarantees like forward secrecy (FS) and post compromise security (PCS). Here, FS means that even if a party gets compromised, previously delivered messages (typically all messages prior to the last key update) remain private. On the other hand, PCS guarantees that even if a party was compromised resulting in full state leakage, normal protocol execution after the compromise ensures that eventually (typically after the next key update) future messages will again be private and authenticated.

Most existing protocols were originally designed for the two party case and do not scale beyond that. Thus, group messaging protocols are usually built on top of a complete network of two party channels. Unfortunately, this means that message sizes (at least for the crucial key update operations) grow linearly in the group size. In view of this, constructing messaging schemes that provide strong security – in particular FS and PCS – while *efficiently* scaling to larger groups is an important but challenging open problem. Designing such a protocol is the ongoing focus of the IETF working group Message Layer Security (MLS) [1].

Instead of constructing a messaging scheme directly, a modular approach where one focuses on the construction of an asynchronous continuous group key agreement (CGKA) protocol seems more natural. CGKA is the multi-party generalisation of the continuous key agreement primitive introduced in [2], and as in the 2-party case, it can be efficiently and generically turned into a group messaging protocol using standard cryptographic primitives. CGKA has been introduced in [3]. Previously, a similar notion for static groups (where one cannot add or remove members after initialisation) has been used in the ART protocol [6] discussed below.

1.1 Continuous Group Key Agreement

Informally, in a CGKA protocol any party ID_1 can initialise a group $\mathsf{G} = (\mathsf{ID}_1, \ldots, \mathsf{ID}_n)$ by sending a message to all group members, from which each group member can compute a shared group key I. The initiator ID_1 must know a public key pk_i of each invitee ID_i, which in practice could be realized by having a key-server where parties can deposit their keys. As this key-management problem is largely orthogonal to the construction of CGKA, in this work we will assume that such an infrastructure exists.

Apart from initialising a group, CGKA allows any party ID_i currently in the group to *update* its key. Informally, after an Update[5] operation the state of ID_i is secure even if its previous state completely leaked to an adversary. Moreover any group member can *add* a new group member, or *remove* an existing group member.

These operations (Update, Add, Remove) require sending a message to all members of the group. As we do not assume that the parties are online at the same time, ID_i cannot simply send a message to ID_j. Instead, all protocol messages are exchanged via an untrusted delivery server. Although the server can always prevent any communication taking place, we require that the shared group key in the CGKA protocol – and thus the messages encrypted in the messaging system built upon it – remains private.

Another issue we must take into account is the fact that (at least for the protocols discussed below) operations must be performed in the same order by all parties in order to maintain a consistent state. Even if the delivery server is honest, it can happen that two parties try to execute an operation at the same time. In this case, an ordering must be enforced, and it is natural to let the delivery server do it. Whenever a party wants to initiate an Update/Remove/Add operation, it sends the message to the delivery server and waits for an answer. If it gets a confirmation, it updates its state and deletes the old one. If it gets a reject, it deletes the new state and keeps the old one. Note that when a party gets corrupted while waiting for the confirmation, both, the old and new state are leaked.

Asynchronous Ratcheting Tree (ART) The first proposal of (a simplified variant of) a CGKA is the Asynchronous Ratcheting Tree (ART) [6]. This protocol (as well as TreeKEM discussed below our new protocol) identifies the group with a binary tree where edges are directed and point from the leaves to the root.[6] Each party ID_i in the group is assigned their own leaf and the leaf is labelled with an ElGamal secret key x_i (known only to ID_i) and a corresponding public value g^{x_i}. The values of internal nodes are defined recursively: an internal node whose two children have secret values a and b has the secret value g^{ab} and public value $g^{\iota(g^{ab})}$, where ι maps group elements to integers. The secret value of the root is the group key. As illustrated in Figure 1, with this setup, a party can update its secret key x to a new key x' by computing a new path from x' to the (new) root, and then send the public values on this new path to everyone in the group so they can switch to the new tree. Note that the number of values that must be shared equals the depth of the tree, and thus (as the tree is balanced) is only logarithmic in the number of parties in the group.

Unfortunately it is not clear how to add or remove parties in the ART protocol, or even initialise it, without at least some of the parties (apart from the initiator of the operation) being online as a party can only update nodes on the path starting at its own leaf.

The authors prove the ART protocol secure even against adaptive adversaries. For the adaptive case, their reduction loses a factor that is super-exponential in the group size. To get meaningful security guarantees based on this reduction requires a security parameter for the ElGamal scheme which is super-linear in the group size, which would result in large messages defeating the whole purpose of using a tree structure.

TreeKEM The TreeKEM proposal [4, 9] is similar to ART as a group is still mapped to a balanced binary tree where each node is assigned a public and secret value. In TreeKEM those values are the public/secret key pair for an arbitrary public-key encryption scheme. As in ART, each leaf is assigned to a party, and only

[5] Throughout we use a capital letter (Update, Add, Remove) if we refer to the operation (not the verb).

[6] The non standard direction of the edges here captures that knowledge of (the secret key of) the source node implies knowledge of the (secret key of the) sink node. Note that nodes therefore have one child and two parents.

this party should know the secret key of its leaf, while the secret key of the root is the group key. Unlike in ART, TreeKEM does not require any relation between the secret key of a node and the secret key of its parent nodes. Instead, an edge $u \to v$ in the tree (recall that edges are directed and pointing from the leaves to the root) denotes that the secret key of v is encrypted under the public key of u. This ciphertext can now be distributed to the subset of the group who knows the secret key of u to convey the secret key of v to them. Below we will refer to this as "encrypting v to u".

To initialise a group, the initiating party creates a tree by assigning the leaves to the keys of the invited parties. She then samples fresh secret/public-key pairs for the internal nodes of the tree and computes the ciphertexts corresponding to all the edges in the tree. (Note that leaves have no ingoing edges and thus the group creator only needs to know the public key.) Finally she sends all ciphertexts to the delivery server. If a party comes online, it receives the ciphertexts corresponding to the path from its leaf to the root from the server, and can then decrypt (as it has the secret key of the leaf) the nodes on this path all the way up to the group key in the root.

As illustrated in Figure 1, this construction naturally allows for adding and removing parties. If ID_i wants to remove ID_j, it simply samples a completely fresh path from a (fresh) leaf to a (fresh) root which replaces the path from ID_j's leaf to the root. It then computes and shares all the ciphertexts required for the parties to switch to this new path *except* the ciphertext that encrypts to ID_j's leaf. If ID_i wants to add ID_j the procedure is similar, ID_i just samples a fresh path which starts at a leaf that is currently not occupied by any party, using ID_j's key as the new leaf node.

Unfortunately, adding and removing parties like this creates a new problem. After ID_i added or removed ID_j, it knowns all the secret keys on the new path (except the leaf). To see why this is a problem, assume ID_i is corrupted while adding (or removing) ID_j (and no other corruptions ever take place), and later – once the adversary loses access to ID_i's state – ID_i executes an Update. Assume we use a naïve protocol where this Update replaces all the keys on the path from ID_i's leaf to the root (as in ART) but nothing else. As ID_i's corruption also leaked keys which are not on this path, and thus were not replaced with this Update, the adversary will potentially still be able to compute the new group key, so Update failed to achieve PCS.

To address this problem, TreeKEM introduced the concept of *blanking*. In a nutshell, TreeKEM wants to maintain the invariant that parties know only the secrets for nodes on the path from their leaf to the root. However, if a party adds (or removes) another party as outlined above, this invariant no longer holds. To fix this, TreeKEM simply declares any nodes with secrets violating the invariant not having any secret (nor public) value assigned to them. Such nodes are called "blanked", and the protocol basically specifies to act as if the child of a blank node is connected directly to the blanked node's parents. In particular, when TreeKEM calls for encrypting something to a blank node, users will instead encrypt to this node's parents. In case one or both parents are blanked one recurses and encrypts to their parents and so forth.

This saves the invariant, but hurts efficiency, as we now no longer consider a binary tree and, depending on the sequence of Adds and Removes, can end up with a "blanked" tree that has effective indegree linear in the number of parties (in particular, this is the case right after a party initialises a group), and thus future Update, Add or Remove operations can require a linear number of ciphertexts to be sent.

The reason one can still hope for TreeKEM to be efficient in practice comes from the fact that blanked nodes can heal: whenever a party performs an Update operation, all the blank nodes on the path from its leaf to the root become normal again. When ID_i adds or removes ID_j all nodes shared by their paths will heal, while all non-shared nodes on the path from ID_j's leaf to the root will be blanked.

1.2 Our Contribution

In this work we propose a new protocol, called Tainted TreeKEM, or simply TTKEM, which is similar to TreeKEM but potentially much more efficient. We also prove a comprehensive security statement for TTKEM which captures the intuition that an Update fixes a compromised state. Our proof can be easily adapted to TreeKEM, for which we can get exactly the same security statement.

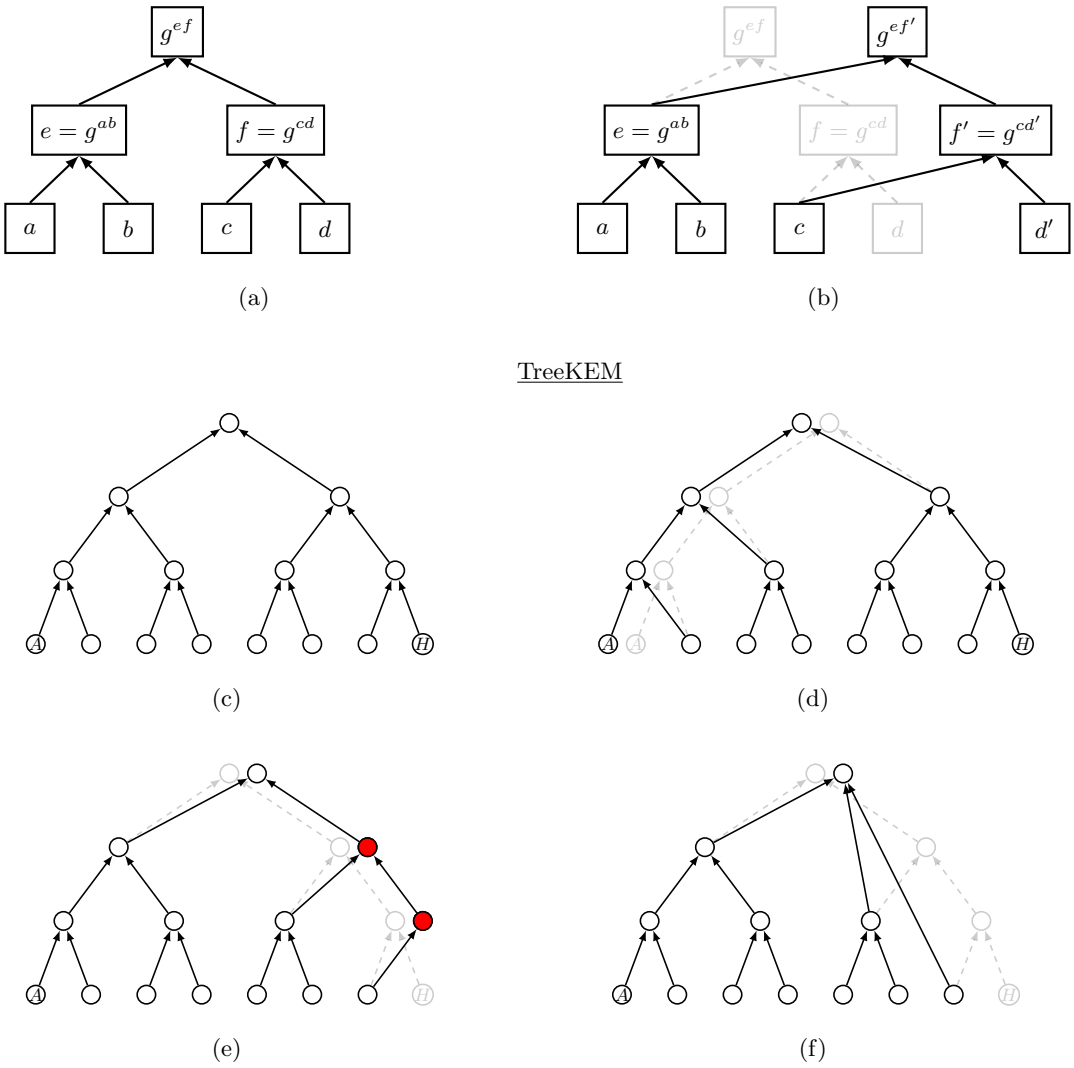


Fig. 1. Top: Illustration of an Update in the ART protocol. The state of the tree changes from (a) to (b) when Dave (node d) updates his internal state to d'. **Bottom**: Update and Remove in TreeKEM and TreeKEM with blanking. The state of a completely filled tree is shown in (c). The state changes from (c) to (d) when Alice (node A) performs an Update operation. This changes to (e) when Alice removes Harry (node H) in naïve TreeKEM (with the nodes that Alice should not know in red) or to (f) in the actual TreeKEM protocol which uses blanking.

It was brought to our attention that a post on the IETF mailing list[7] from February 2018 suggested how to augment the ART protocol with Add and Remove operations in a way which is very similar to the tainted approach we take for TreeKEM. As the construction itself is not completely original, we consider the security statement and its proof as our main contribution.

Tainted TreeKEM (TTKEM) As just outlined, the reason TreeKEM can be inefficient comes from the fact that once a node is blanked, we cannot simply encrypt to it, but instead must encrypt to both its

[7] [MLS] Removing members from groups Jon Millican <jmillican@fb.com> Mon, 12 February 2018 11:01 UTC https://mailarchive.ietf.org/arch/msg/mls/4-gvXpc-LGbWoUS7DKGYG65lkxs

parents, if those are blanked, to their parents, and so forth. This process must stop at the leaves as they cannot get blanked.

The rationale for blanking is to enforce the invariant that the secret keys of (non-blanked) nodes is only known to parties whose leaves are ancestors of this node. This seems overly paranoid, assume Alice removed Henry as illustrated in Figure 1, then the red nodes must be blanked as Alice knowns their value, but it is instructive to analyse when this knowledge becomes an issue if no blanking takes place: If Alice is not corrupted when sending the Remove operation to the delivery server there is no issue as she will delete secret keys she should not know right after sending the message. If Alice is corrupted then the adversary learns those secret keys. But even though now the invariant doesn't hold, it is not a security issue as an adversary who corrupted Alice will know the group key anyways. Only once Alice updates (by replacing the values on the path from her leaf to the root) there is a problem, as without blanking not all secret keys known by the adversary are replaced, and thus he will be able to decrypt the new group key while an Update should have fixed that (more generally, we want the group key to be safe once all the parties whose state leaked have updated).

Keeping dirty nodes around: tainting versus blanking. In TTKEM we use an alternative approach, where we don't blank nodes, but instead keep track of which secret keys of nodes have been created by parties who are not supposed to know them. Specifically, we call nodes whose secret keys were created by a party ID_i which is not a ancestor of the node as *tainted (by* ID_i*)*. The group keeps track of which nodes are tainted and by whom.

A node tainted by ID_i will be treated like a regular node, except for the case when ID_i performs an Update or is being removed, in which case we require that the tainted node gets updated.

Let us remark that tainted nodes can heal similarly to blanked nodes in TreeKEM: once a party performs an Update, all nodes on the path from its leaf to the root are no longer tainted.

Efficiency of TTKEM vs TreeKEM. Efficiency wise TreeKEM and TTKEM are incomparable. Depending on the sequence of operations performed either TreeKEM or TTKEM can be more efficient (or they can be identical). Thus, which one will be more efficient in practice will depend on the distribution of operation patterns we observe. We make the case that for some natural cases TTKEM will significantly outperform TreeKEM.

Consider the (insecure) "naïve TreeKEM" protocol which is TreeKEM without blanking, and let us compare its efficiency for a sequence of operations with TreeKEM and TTKEM. Compared to naïve TreeKEM, in TreeKEM, a blanked node creates extra work (i.e., requires more ciphertexts to be created and distributed) whenever a party must encrypt to the blanked node (recall this means encrypting using this node's public-key). On the other hand, in TTKEM we get some extra work to do (compared to naïve TreeKEM but also TreeKEM) whenever some party ID_i updates or is being removed for which there are nodes tainted by ID_i in the tree, as now all those nodes must be updated too.

Summing up, in TreeKEM a blanked node creates extra work (compared to naïve TreeKem) for *every operation that needs to encrypt to the blanked node*, while in TTKEM a node tainted by ID_i creates extra work *every time* ID_i *updates or is being removed.*

When we compare the efficiency of the CGKA protocols we focus on the number of ciphertexts a party must exchange with the delivery server for an (Update, Add or Remove) operation. The reason to focus on the efficiency of the group member initiating an operation, and not the efficiency of the group members who need to process this operation, is justified by the fact that in all protocols considered, a group member who needs to process an operation just needs to download and decrypt a logarithmic number of ciphertexts (or even just a single one if the keys on a path are all derived from a single seed), so the protocols don't differ in this aspect and efficiency wise there is not much left to improve.[8]

Although the efficiency of TreeKEM and TTKEM are incomparable, it is clear that TTKEM will be more efficient than TreeKEM whenever we have just a small subset of parties who perform most of the Add

[8] But let us mention that there is still room for improvement in the case where a group member comes online and must process a large number of operations as these could potentially be somehow batched by the server.

and Remove operations, but only rarely perform Updates themselves. In practice, this could correspond to a setting where we have a small group of administrators who are the only parties allowed to add/remove parties. The efficiency gap grows further if the administrators have a lower risk of compromise than other group members and thus can be required to update less frequently. In this setting, TTKEM approaches the efficiency of naïve TreeKEM. TreeKEM on the other hand can preform much worse, in particular if the non-administrators rarely update, and thus the blanked nodes on their path don't heal.

Security of (Tainted) TreeKEM A main contribution of this work is a security proof for TTKEM for a *comprehensible* security statement that intuitively captures how Updates ensure forward and post compromise security, in a *strong* security model. This security statement and its proof can also be adapted to TreeKEM, we thus also give the first formal security statement and proof for TreeKEM. In particular, the adversary we consider is

- Fully adaptive: it can instruct any party to initialize an Init/Update/Add/Remove operation and corrupt parties completely adaptively based on the transcript so far. While a party is corrupted its entire state (secret-keys, randomness used) is visible to the adversary.
- Partially active: it has full control over the delivery server and can send incoherent messages to various parties (e.g. inform a party that its Update has been rejected while ordering other group members to perform this Update, thus leaving the group members in an inconsistent state).
  What we do not allow the adversary to do is to create ciphertexts itself (from scratch or by mauling learned ciphertexts).

The goal of the adversary is to break the security of a target group key that was, at some point in the execution, considered to be the current group key by at least one group member, and that given the actions taken by the adversary so far is not trivially insecure. This means this key cannot be trivially decrypted from ciphertexts observed so far using secret keys leaked by corrupted parties.

We define an intuitive predicate that specifies for which group keys this is the case. Deciding whether this predicate holds can be determined by just looking at the transcripts (including corruption queries) of the individual group members and not some complicated global structure like the relative position of parties in the tree. Having such a simple predicate is important as we want a security notion which has a simple intuition behind it and in particular clearly captures forward secrecy and post compromise security.

The predicate becomes particularly simple if we assume the adversary never forces the group members into an inconsistent state (i.e., always either all or no party is instructed to process an instruction). In this case the predicate holds if no group member was corrupted in a window between two Updates in which the group key falls.

1.3 The adversarial model.

We anticipate an adversary who works in rounds, in each round he can adaptively choose an action, including start/stop corrupting a party, instruct a party to initalize an operation or to relay a message, a more detailed description is below.

The adversary can choose to corrupt any party, after which its state becomes visible to the adversary. He can also choose to stop the corruption of a currently corrupted party.

The adversary can instruct a party to initalize an Init/Update/Remove/Add operation. This party then immediately outputs the corresponding message to be sent to the delivery server.

The adversary has complete control over the delivery server, and thus the scheduling of the messages. In particular, we do not assume that the delivery server enforces an ordering of operations as an honest server would. The adversary can even mess with a single operation: Recall that the honest delivery server, upon receiving a message from ID_i issuing an operation, will either send a reject to ID_i, or will send a confirmation to ID_i (who will then process the operation) and relay the (relevant parts of the) message to the group members (who will process the operation). Our adversary could send a reject to ID_i, while still relaying the message to a party ID_j who will process it (while ID_i thinks the operation was rejected).

Once two parties are in an inconsistent state (this basically means the sequences of operations processed so far by the two parties are distinct, and none is a prefix of the other), they will never be able to synchronise again, but as said, if the delivery server is malicious (in particular, doesn't enforce an ordering), then we cannot hope for correctness, but we still achieve security in this case.

We assume that a party, after receiving a message from the server asking to process an operation, will only do so if this operation was initiated by a party that (when triggering this operation) had the same view of the state of the group as itself, but this can easily be enforced by adding a (collision-resistant) hash of the operations processed so far [7, 11].[9]

One thing we do not allow the adversary to do is to create ciphertexts itself (either from scratch or by mauling a received ciphertext). This assumption is required for our reduction to go through, but we don't see an attack if we drop it. Proving security against fully active adversaries is an interesting open problem.

The goal of the adversary is to break the security of a group key (i.e., a secret key that is contained in the root in the view of at least one party) that – given the sequence of actions performed – it should not trivially know.

The reduction. We reduce the security of the protocol to the security of the underlying encryption scheme. By using the framework of Jafargholi et al. [10] we achieve a quasipolynomial security loss (in the order of $q^{\log\log n}$, where n is an upper bound on the number of group members and q is the number of Init/Update/Remove/Add queries the adversary issues) even when considering an *adaptive* adversary who can choose every action adaptively depending on the previous messages he observed. This should be contrasted with the security bound for ART (cf. subsection 1.1) – the only published security proof for a CGKA like protocol considering fully adaptive adversaries – which loses a superexponential factor.

If we want to get security guarantees in practice from a reduction with exponential (in the group size n) security loss this requires a security parameter that is linear in n for the underlying encryption scheme, which in turn will result in ciphertexts (which encrypt secret keys) of size linear in n. But such large packet sizes defeat the whole purpose of using a tree structure in the first place, we could as well use pairwise channels. In contrast, a security loss in the order of $q^{\log\log n}$ as we get it only requires the underlying security parameter to be of size $\log(q^{\log\log n}) = \log q \cdot \log\log n$ which can be practical.

The GSD game. Concretely, we observe that the security of TTKEM can be cast as a restricted version of the generalised selective decryption [12] (GSD) game. There, an adversary gets a set of public keys $pk_1, \ldots, pk_n$ (but not the corresponding secret keys $sk_1, \ldots, sk_n$) and can then ask for encryptions of secret keys, e.g. a query (i, j) would return the ciphertext $Enc_{pk_i}(sk_j)$ of sk_j under pk_i. We think of the pk_i's as nodes in a graph, and add a directed edge $i \to j$ when the adversary makes a query $Enc_{pk_i}(sk_j)$. The adversary can also make corruption queries, where on input i he gets sk_i. The adversary can then challenge a public key pk_i, but to be a valid challenge it must hold that (1) pk_i is a sink, (2) none of its ancestors is corrupted and (3) the entire graph is acyclic. The goal of the adversary is to break the security of this challenge key (i.e., distinguish $Enc_{pk_i}(m_0)$ from $Enc_{pk_i}(m_1)$ for m_0, m_1 of its choice).

It is not hard to see that the TTKEM (and also TreeKEM) security experiment can be viewed as a GSD game, where (assuming an upper bound of n on group size) the graph will be of depth at most $\lceil\log(n)\rceil$ and assuming the adversary makes at most q operations, will have at most $|V| \leq n \cdot q$ nodes (if we count initialization as n queries). A standard hybrid argument can be used to prove that no *selective* adversary can win the GSD game with advantage $|V| \cdot \epsilon \leq n \cdot q \cdot \epsilon$ assuming he cannot break security of the underlying PKE with advantage ϵ. Here, selective means the adversary must commit to all its queries before getting to see any queries or even getting the public keys. Proving security for an adaptive adversary is much more challenging, a standard complexity leveraging argument would loose an exponential (in $|V| \approx n \cdot q$) factor. The works of [12, 8] improve this to quasipolynomial in the depth of the graph. Of course we are interested in adaptive security of TTKEM. Just applying the framework would give us $\approx (q \cdot n)^{\log(n)} \cdot \epsilon$ security, which is already much better than an exponential loss. This bound only relies on the size and depth of the underlying graph.

[9] For efficiency reasons one could use a Merkle hash so that from the hash of a (potentially long) string T we can efficiently compute the hash of T concatenated with a new operation t.

Taking the more restrictive structure of the queries and the graph constructed in the TTKEM security game into account we can significantly improve this to an only slightly superpolynomial $\approx q^{\log\log(n)} \cdot \epsilon$.

The safe predicate. We still need to understand what the security in the GSD games implies for the TTKEM security game. In particular, we must translate what it means to be a valid challenge in GSD (recall this means the challenged key must be a sink node with no corrupted ancestors) to challenges on a group key in TTKEM.

One could simply define the TTKEM security by saying that a group key is a valid challenge if it is a valid challenge in the underlying GSD game, but this would not be very intuitive and thus useful as a security guarantee, in particular, it would require to take into account where in the tree individual keys were added. Instead, we will define an intuitive predicate in the TTKEM game, and show that a group key is safe with respect to this predicate if it is a valid challenge in the GSD game.

1.4 Asynchronous Continuous Group Key Agreement Syntax

Definition 1. *(Asynchronous Continuous Group Key Agreement)*
An asynchronous continuous group key agreement *(CGKA) scheme is an* 8*-tuple of algorithms* $\mathsf{CGKA} = (\mathsf{keygen}, \mathsf{init}, \mathsf{add}, \mathsf{rem}, \mathsf{upd}, \mathsf{dlv}, \mathsf{proc}, \mathsf{key})$ *with the following syntax and semantics:*

KEY GENERATION**:** *Fresh InitKey pairs are generated using* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{keygen}(1^\lambda)$*. They are used to invite parties to join a group.*

INITIALIZE A GROUP**:** *For* $i \in [2, n]$ *let* pk_i *be an InitKey PK belonging to party* ID_i*. Let* $\mathsf{G} = (\mathsf{ID}_1, \ldots, \mathsf{ID}_n)$*. Party* ID_1 *creates a new group with membership* G *by running:*

$$(\gamma, [W_2, \ldots, W_n]) \leftarrow \mathsf{init}\,(\mathsf{G}, [\mathsf{pk}_1, \ldots, \mathsf{pk}_n])$$

and sending welcome message W_i *for party* ID_i *to the server. Finally,* ID_1 *stores its* local state γ *for later use.*

ADDING A MEMBER**:** *A group member with local state* γ *can add party* ID *to the group by running* $(\gamma', W, T) \leftarrow \mathsf{add}(\gamma, \mathsf{ID}, \mathsf{pk})$ *and sending* welcome message W *for party* ID *and the* add message T *for all group members (including* ID*) to the server. He stores the old state* γ *and new state* γ' *until getting a confirmation from the delivery server as defined below.*

REMOVING A MEMBER**:** *A group member with local state* γ *can remove group member* ID *by running* $(\gamma', T) \leftarrow \mathsf{rem}(\gamma, \mathsf{ID})$ *and sending the* remove message T *for all group members (including* ID*) to the server and storing* γ, γ'*.*

UPDATE**:** *A group member with local state* γ *can perform an update by running* $(\gamma', T) \leftarrow \mathsf{upd}(\gamma)$ *and sending the* update message T *for all group members to the server and storing* γ, γ'*.*

CONFIRM AND DELIVER**:** *The delivery server upon receiving a (set of) CGKA protocol message(s)* T *(including welcome messages) generated by a party* ID *by running* $\mathsf{dlv}(\mathsf{ID}, T)$ *either sends* T *to the corresponding member(s) and sends a message* $\mathsf{confirm}$ *to* ID*, in which case* ID *deletes it's old state* γ *and replaces it with* γ'*, or sends a message* reject *to* ID*, in which case* ID *deletes* γ'*.*

PROCESS**:** *Upon receiving an incoming (set of) CGKA protocol message(s)* T *(including welcome messages) a party immediately processes them by running* $(\gamma, I) \leftarrow \mathsf{proc}(\gamma, T)$*.*

GET GROUP KEY**:** *At any point a party can extract the current group key* I *from its local state* γ *by running* $(\gamma, I) \leftarrow \mathsf{key}(\gamma)$*.*

Intuitively, updates serve to refresh all parts of the joint group state held by the party doing the updating. This has the effect to (hopefully) make any part of that party's local state which has previously leaked useless. In particular, this is the primary mechanism through which PCS is achieved.

We remark that while the protocol allows any group member to add a new party to the group as well as remove any member from the group it is up to the higher level message protocol (or even higher level application) to decide if such an operation is indeed permitted. (If not, then clients can always simply choose to ignore the add/remove message.) At the CGKA level though all such operations are possible.

2 Description of TTKEM

2.1 Notation

In this work, a directed binary tree $\mathcal{T}$ is defined recursively as a graph that is either the empty graph, a root node, or a root node whose parents are root nodes of trees themselves. Note that this corresponds to a standard definition of trees with reversed edges. We choose this definition of trees since it is much more intuitive in our context and highlights the connection between the protocol and the GSD game used for the security proof (cf. Section 6). Note that paths in the tree now start at leaves and end at the root node. Throughout this section, for simplicity of notation we assume an implicit user state γ, which contains a directed binary tree $\mathcal{T}$ with unique sink v_{root}. It maintains public keys associated to each node and user ID's associated to each leaf node. Throughout the remaining document we will use the functions `child`, `parents`, `partner` to refer to the child, parents and partner (the other parent of the child) of any given node. The function `index`(ID) returns the leaf ID has assigned, and `get_pk`, `get_sk`, `get_tainter` the public key, secret key and tainter ID of a given node respectively. Similarly, the binary functions `set_pk`(v_i, pk_i), `set_sk`(v_i, sk_i) and `set_tainter`(v_i, ID) overwrite the public key, secret key or tainter ID associated to v_i. We will use the function `path` to recover the nodes in the path of a user ('s leaf) to the root. Finally we use `get_members`(), `get_tree`(), `get_hash`() to recover the member list, tree or transcript hash from a state; To update one's view of group state, we use the functions `add_party`(ID, pk) to add ID to the leftmost free spot in the tree; `remove_party`(ID) to remove ID; `update_hash`(T) to update our transcript hash with the message T ; `init_state`$(\mathcal{M}, \mathcal{T}, \mathcal{H})$ to initialize our state after joining; and `update_pks_and_tainter`$(new\_pks, \mathsf{ID}, \mathsf{ID}')$ to update the public keys of nodes corresponding to ID, and changing their tainter ID to ID'.

To achieve FS and PCS it is necessary to constantly renew the secret keys used in the protocol. We will do this through group operations like **Update** or **Remove**. To avoid confusion, whenever we are referring to the renewal of a particular key or set of keys (as opposed to the generic group operation), we will use the term *refresh*. When clear, we will use the term update to refer to any group operation that prompts us into a new state.

2.2 Overview

The protocol uses a directed binary tree $\mathcal{T}$ as an underlying structure. The nodes in the tree are associated with the following values

- a seed Δ
- (all nodes except the root) a secret/public key pair derived deterministically from the seed

$$(pk, sk) \leftarrow KeyGen(\Delta)$$

- (only leaf nodes) a credential
- (all except leaves and root) a tainter ID

The root has no public/secret key pair associated with it, instead its seed is the current group key. The epoch key (used for a symmetric authenticated encryption scheme to encrypt the exchanged messages) for the group messaging protocol built on top of this continuous group key agreement protocol is determined by the current group key and the previous epoch key. Each group operation will refresh a part of the tree, always including the root and thus resulting in a new group key which can be decrypted by all members of the current group.

Each user should have a consistent view of the public information in the tree, namely public keys, credentials, tainter IDs and past operations. Furthermore, group members will have a partial view of the seeds (and thus the secret keys).

More precisely, every user has an associated *protocol state* $\gamma(\mathsf{ID})$ (or state for short when there is no ambiguity), which represents everything users need to know to stay part of the group (we implicitly assume a particular group, considering different groups secrets independent). In particular, we define a state as the triple $\gamma(\mathsf{ID}) = (\mathcal{M}, \mathcal{T}, \mathcal{H})$, where:

– $\mathcal{M}$ denotes the set of group members, i.e. the set of ID's that are part of the group
– $\mathcal{T}$ denotes a binary tree defined as above, where for each group member, their credential is associated to a leaf node.
– $\mathcal{H}$ denotes the hash of the group transcript so far. This is to ensure consistency.

Each user also has a *pending state* $\gamma'(\mathsf{ID})$ which stores the changes done by the last operation add, rem or upd while they wait for it to be confirmed or rejected.

As stated above, a user will generally not have knowledge of the secret keys associated to all tree nodes. In fact, we would ideally like the following predicate to hold:

A user knows the secret key associated to a node if and only if that node is in the path from that user's leaf to the tree root.

This, however, does not seem possible to guarantee if we do not want our efficiency to degrade: if we want to add a new potentially offline member to our group while keeping the binary tree structure, we need to communicate to them the secret keys along their path to the root. However, if we do not already know them, i.e. our leaves are not partners, how should we do this? The problem is present also in the case of removals. Recall the case outlined in the introduction: Alice wants to kick Harry out of the group. She will need to change the secrets in the nodes on his path to the root, without Harry learning the new secrets. The natural approach would be for Alice to sample a fresh path for Harry and deliver the appropriate secrets to everyone but Harry. However, as with adding, Alice would need to somehow obliviously sample them to not learn the secrets - something not a priori easy while at the same time being able to calculate the corresponding public keys, communicate the secrets to the different parties, etc. TreeKEM ensures the validity of the predicate by blanking the problematic nodes. Instead, the approach we take is to allow the predicate to be violated. We observe that this will not be a problem as long as we have a mechanism to keep track of those nodes and refresh them when necessary, towards this end we introduce the concept of tainting.

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

– After the initialization of the group, all nodes are tainted by the group creator (except leafs and the creator's path) .
– Whenever a party ID_i updates, the refreshed nodes on the path from ID_i to the root become untainted.
– Whenever a party ID_i updates, all refreshed nodes *not* on the path from ID_i to the root become tainted with tainter ID ID_i.

Hierarchical derivation of updates. When refreshing a whole path, instead of sampling a new secret for each node, we sample a seed Δ_0 and derive all the secrets for that path from it. This way, we reduce the number of decryptions other parties will need to perform to process the update, as parties only need to recover the seed for the "lowest" node that concerns them, and then can derive the rest locally. To derive the different new secrets we use two hash functions H_1, H_2 together with a $KeyGen$ function that outputs a secret-public key pair, following the specification of the MLS draft [9], where more details can be found.

$$\Delta_{i+1} := H_1(\Delta_i)$$
$$(sk_i, pk_i) \leftarrow KeyGen(H_2(\Delta_i))$$

where Δ_i is the seed for the ith node (the leaf being the 0th node, its child the 1st etc.) on the path and (sk_i, pk_i) its new key pair.

With the introduction of tainting, it is no longer the case that all nodes to be refreshed lie on a path. What we propose is to cover all the nodes to be refreshed with the minimal number of paths and use a different seed for each path. For this we need a unique path cover, as users processing the update will need to know which nodes secrets depend on which.

Formally, for a user id, we want a set of paths $P_i = \{v_{i,0}, \ldots, v_{i,m_i}\}$ such that:

- $\texttt{child}(v_{i,j}) = v_{i,j+1}$ *(P_i is a path)*
- $v_{i,j} \neq v_{k,l}$ if $i \neq k \vee j \neq l$ *(each node is only in one path)*
- $\texttt{get\_tainter}(v_{i,0}) = id$ *(the start of each path is a node tainted by id)*
- $\forall v \in P_j : \texttt{child}(v) \neq v_{i,0}$ *(paths are maximal)*
- $P_i \bigcap P_{id} = \emptyset$ *(paths are disjoint from main path to root)*
- $\texttt{child}(v_{i,m_i}) \in P_{id} \vee \texttt{child}(v_{j,m_j}) \in P_i$ with $i < j$ *(the partition is unique)*
- $v_{i,0} < v_{j,0}$ if $i < j$ *(there is a total ordering on paths)*

where P_{id} is the path from the user's leaf to the root and $v_i < v_j$ if v_i is more to the left. We denote this ordered partition by $\texttt{tainted-by}(id)$. Note that the first five conditions ensure that the partition contains only the nodes to be refreshed and that its size is minimal, while the sixth and seventh conditions guarantee that the partition is unique. A common ordering of the paths is needed, since when we refresh two paths that "intersect" (such that $\texttt{child}(v_{i,m_i}) \in P_j$, as the blue and red paths in the image below for example), the node secret in the "upper" path (the red path in this example) needs to be encrypted under the *new* public key of the node in the "lower" path (the new blue node) to achieve PCS. Thus, in this case, the blue path will need to be refreshed before the red one when processing the update. In general we will refresh paths right to left, i.e. P_i will be refreshed after P_j if $i < j$.

Let us stress that a party processing an update that involves tainted nodes might need to retrieve and decrypt more than one encrypted seed from the delivery server as the refreshed nodes on its path are not all derived hierarchically. Though even in the worst case no party needs to decrypt more than $\log n$ ciphertexts

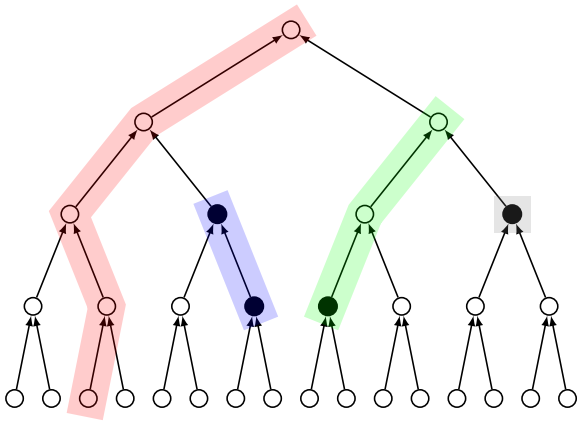


Fig. 2. Schematic diagram showing the path partition for an update done by Charlie (3-rd leaf node). The black nodes are the ones tainted by him. The order of paths would thus be blue, green, grey; with a user refreshing them in the opposite order, followed by Charlie's path to the root.

2.3 TTKEM Dynamics

Whenever a user wants to perform a group operation, she will generate the appropriate Update, Add or Remove message and send it to the delivery server, which will then respond with a confirm or reject. If the (honest) delivery server confirmes an operation it will also deliver it to all the group members, who will

process it and update their states accordingly. The initiator of a group operation creates a message T which contains all information needed by the other group members to process it (though different members might only need to retrieve a part of T for performing the update) and in case of an Add also a welcome message W for the new member. The message T contains the following fields:

- T_{sender} - ID of the sender
- T_{op} - type of operation (remove/add/update)
- T_{new_seeds} - vector of ciphertexts which contains the encrypted seeds under the appropriate keys of all refreshed nodes
- T_{new_pks} - vector of new public keys (derived from the new seeds) for all refreshed nodes
- $T_{\mathcal{H}}$ - hash-transcript

If the operation is a removal, the ID of the party removed will also be included in T_{op}. Similarly, in Add messages, T_{op} will contain the ID of the party added, together with the public key used to add him. A welcome message W would also contain the type of operation (*welcome*) and the sender ID, but additionally include:

- W_{seed} - an encryption of the child node's seed
- $W_{\mathcal{T}}$ - the current tree structure, with public keys
- $W_{\mathcal{M}}$ - current list of group members
- $W_{\mathcal{H}}$ - current hash-transcript of the group

A new member should also be communicated the current symmetric epoch key used to communicate text messages. As this is not strictly part of the GCKA we ignore it for simplicity.

In order to refresh the node secrets we use the function *refresh*(γ, ID, T), which takes a user's state, a user in the group and a message T. It generates new secrets for all the nodes in that user's path to the root as well as all nodes tainted by them, update γ accordingly and store their encryptions in T_{new_seeds}.

refresh (γ, ID, T)
  $P_0 \leftarrow \gamma.\texttt{path}(\mathsf{ID})$
  $\{P_1, \ldots, P_n\} \leftarrow \gamma.\texttt{tainted-by}(\mathsf{ID})$ *#refresh all paths from tainted nodes to root*
  for $i = n, \ldots, 0$ **do**
    $v_{i,0}, \ldots, v_{i,m} \leftarrow P_i$
    $\{\Delta_{i,0}, \ldots \Delta_{i,m}\} \leftarrow \texttt{expand}(\texttt{gen-seed}(), m+1)$
    for $p \in \textit{parents}(v_{i,0})$ **do**
      #encrypt first to parents of 1st node
      if $p \neq \perp$ **then**
        $T_{new\_seeds}.insert(Enc_{\gamma.\texttt{get\_pk}(p)}\Delta_{i,0})$
    refresh-node$(\gamma, v_{i,0}, \Delta_{i,0}, T)$
    for $j = 1, \ldots, m$ **do**
      if $\gamma.\textit{partner}(v_{i,j-1}) \neq \perp$ **then**
        $T_{new\_seeds}.insert(Enc_{\gamma.\texttt{get\_pk}(\gamma.\texttt{partner}(v_{i,j-1}))}\Delta_{i,j})$
      refresh-node$(\gamma, v_{i,j}, \Delta_{i,j}, T)$

We use the function *refresh-node* that inputs a user local state γ, a node v, a seed Δ and message T. It updates the information related to v in the state γ using Δ to derive the new public and secret key and store the public key in T_{new_pks}.

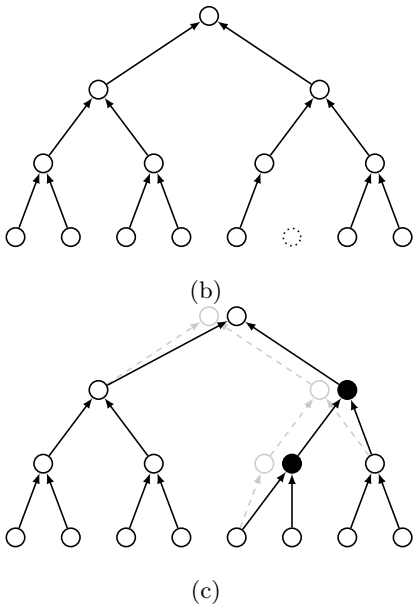
refresh-node (γ, v, Δ, T)
  if $v = v_{root}$ **then**
    $\gamma.\texttt{set\_sk}(v_{root}, \Delta)$
  else
    $(sk, pk) \leftarrow KeyGen(H_2(\Delta))$
    $\gamma.\texttt{set\_pk}(v, pk)$; $\gamma.\texttt{set\_tainter}(v, \texttt{me})$
    $T_{new\_pks}.insert(pk)$
    if $v \in \gamma.\textit{path}(\mathsf{ID})$ **then**
      $\gamma.\texttt{set\_sk}(v, sk)$

Initialize. To create a new group with parties $\{\mathsf{ID}_1, \ldots, \mathsf{ID}_n\}$, a user ID_1 generates a new tree where the leaves correspond to the parties of the group (including themselves), with associated public keys the ones used to add them. The group creator then samples new key pairs for all the other nodes in the tree (optimizing with hierarchical derivation) and crafts welcome messages for each party.

Add. To add a new member to the group, Alice identifies a free spot for them (for consistency, the left-most free spot), hashes her secret key together with some freshly sampled randomness to obtain a seed Δ, and derives seeds for the path to the root, overwriting the previous ones. She then encrypts the new seeds to all the nodes in the path (one ciphertext per node suffices given the hierarchical derivation). The reason for such a derivation of Δ is that the new keys will be secure against an adversary that does not have knowledge of Alice's secret key or control/knowledge of the randomness used. We use the pointer `me` to refer to the identity of the user sending the protocol message and h to refer to the hash function used.

add $(\gamma, \mathsf{ID}, pk)$
- $\gamma' \leftarrow \gamma$
- γ'.`add_party`(ID, pk)
- $\{v_0, \ldots, v_d\} \leftarrow \gamma'$.`path`$(\mathsf{ID})$
- $sk \leftarrow \gamma'$.`get_sk`$(\gamma'$.`index`$(\texttt{me}))$
- $r \leftarrow \$; \Delta \leftarrow h(sk, r)$
- $\{\Delta_0, \ldots, \Delta_d\} \leftarrow$ `expand`$(\Delta, d+1)$
- *refresh-node*$(\gamma', v_0, \Delta_0, T)$
- **for** $i = 1, \ldots, d$ **do**
  - $u \leftarrow \gamma$.`partner`(v_{i-1})
  - **if** $u \neq \perp$ **then**
    - $T_{new\_seeds}.insert(Enc_{\gamma.\texttt{get\_pk}(u)}\Delta_i)$
  - *refresh-node*$(\gamma, v_i, \Delta_i, T)$
- $T_{op} \leftarrow (add, \mathsf{ID}, pk)$
- $T_{sender} \leftarrow \texttt{me}$
- $T_{\mathcal{H}} \leftarrow \gamma$.`get_hash`$()$
- γ'.`update_hash`(T)
- $W_{op} \leftarrow welcome$
- $W_{sender} \leftarrow \texttt{me}$
- $W_{seed} \leftarrow Enc_{pk}(\Delta)$
- $W_{\mathcal{T}} \leftarrow \gamma'$.`get_tree`$()$
- $W_{\mathcal{H}} \leftarrow \gamma$.`get_hash`$()$
- $W_{\mathcal{M}} \leftarrow \gamma$.`get_members`$()$
- **return**(γ', W, T)

(a)



(b)

(c)

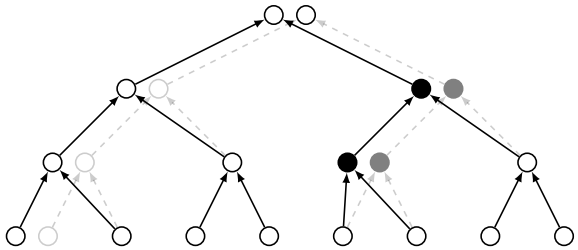
Fig. 3. (a) Pseudocode for the Add operation. Sample Add operation: (b) illustrates the state of the tree before Alice adds Frank after which it turns into (c).

Update. To perform an Update, a user refreshes the nodes in its path to the root and also all the nodes tainted by him. We do this using the function *refresh*, adding information about the type of operation (*upd*) and the initiator of that operation `me`.

Remove. To remove a user j, user i performs an Update on behalf of j, refreshing all the nodes in j's path to the root as well as all nodes tainted by j (which will now become tainted by i). As with updates, we do this by calling the function *refresh*, adding information about the type of operation and the initiator of that operation. Note that a user cannot remove itself. Instead, we imagine a user that wants to leave the group could request for someone to remove him and delete his state.

upd (γ)
- $T_{op} = upd$
- $T_{sender} = \texttt{me}$
- $T_{\mathcal{H}} = \gamma.\texttt{get\_hash}()$
- $\gamma' \leftarrow \gamma$
- $refresh(\gamma', \texttt{me}, T)$
- $\gamma'.\texttt{update\_hash}(T)$
- **return** (γ', T)

(a)

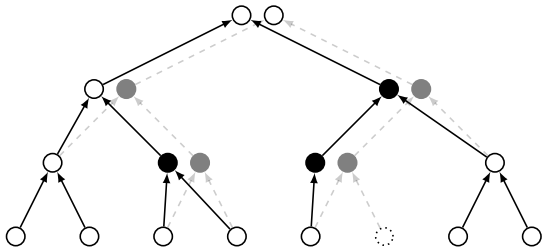


(b)

Fig. 4. (a) Pseudocode for the Update operation. (b) A sample Update operation: Alice added Eve to the group which resulted in the tainted nodes (filled). Alice decided to later update herself. The state of the tree before the Update is in a lighter shade.

rem (γ, ID)
- **req** $\texttt{me} \neq \mathsf{ID}$
- $T_{op} = (rem, \mathsf{ID})$
- $T_{sender} = \texttt{me}$
- $T_{\mathcal{H}} = \gamma.\texttt{get\_hash}()$
- $\gamma' \leftarrow \gamma$
- $refresh(\gamma', \mathsf{ID}, T)$
- $\gamma'.\texttt{remove\_party}(\mathsf{ID})$
- $\gamma'.\texttt{update\_hash}(T)$
- **return** (γ', T)

(a)



(b)

Fig. 5. (a) Pseudocode for the Remove operation. (b) Alice removes Frank (dotted) and in the process has to update his tainted nodes. Old state is again showed in gray.

For our process algorithm we use the algorithms *getEncryption*, *update-path* and *proc-refresh* as subroutines. The function *getEncryption* inputs a user local state γ, a node v_0, a set of paths P_i and the set of encryptions received from the Update/Remove message, and returns the encryption corresponding to v_0. Given path P, seed Δ, and update author ID, *update-path* updates P using Δ as seed, and adding ID as tainter of nodes along P. Finally, *proc-refresh* takes a user ($\texttt{me}$) local state γ, the set of encryptions received from the Update/Remove message T_{new_seeds}, the id ID of the user that made the update/was removed, and the user *sender* that made the operation (distinct from ID if the operation was a Remove), and it updates all the secret keys in the path from the $\texttt{me}$ leaf to v_{root}.

update-path $(\gamma, P, \Delta, \mathsf{ID})$
- **for** $v \in P$ **do**
  - **if** $v = v_{root}$ **then**
    - $\gamma.\texttt{set\_sk}(v_{root}, \Delta)$
  - **else**
    - $(sk, \_) \leftarrow KeyGen(H_2(\Delta))$
    - $\Delta \leftarrow H_1(\Delta)$
    - $\gamma.\texttt{set\_sk}(v, sk)$

process-refresh $(\gamma, T_{new\_seeds}, \mathsf{ID}, sender)$
- $P_0 \leftarrow \gamma.\texttt{path}(\mathsf{ID})$
- $\{P_1, \ldots, P_n\} \leftarrow \gamma.\texttt{tainted-by}(\mathsf{ID})$ *#refresh all paths from tainted nodes to root*
- **for** $i = n, \ldots, 0$ **do**
  - $\{v_0, \ldots, v_n\} \leftarrow \texttt{intersection}(P_i, \gamma.\texttt{path}(\texttt{me}))$
  - $enc \leftarrow getEncryption(\gamma, v_0, P_0, T_{new\_seeds})$
  - $(p_l, p_r) \leftarrow \gamma.\texttt{parents}(v_0)$
  - **if** $p_l \neq \bot \wedge p_l \in \gamma.\mathit{path}(\mathit{me})$ **then**
    - $sk \leftarrow \gamma.\texttt{get\_sk}(p_l)$
  - **else**
    - $sk \leftarrow \gamma.\texttt{get\_sk}(p_r)$
  - *update-path*$(\gamma, \{v_0, \ldots, v_n\}, Dec_{sk}(enc), sender)$

Process. When a user receives a protocol message T, it identifies which kind of message it is and performs the appropriate update of their state. Updates and Removes are processed using the *proc-refresh* algorithm; additions are processed using the *update-path* algorithm. If it is a confirm or a reject it updates the current local state accordingly and remove the information in the pending local state.

process (γ, T)
- **req** $T_{\mathcal{H}} = \gamma.\texttt{get\_hash}()$
- **if** $T_{op} = upd$ **then**
  - *proc-refresh*$(\gamma, T_{new\_keys}, T_{sender}, T_{sender})$
  - $\gamma.\texttt{update\_pks\_and\_tainter}(T_{new\_pks}, T_{sender}, T_{sender})$
- **if** $T_{op} = (rem, \mathsf{ID})$ **then**
  - **if** $\mathsf{ID} \neq \mathit{me}$ **then**
    - *proc-refresh*$(\gamma, T_{new\_keys}, \mathsf{ID}, T_{sender})$
    - $\gamma.\texttt{update\_pks\_and\_tainter}(T_{new\_pks}, \mathsf{ID}, T_{sender})$
    - $\gamma.\texttt{remove\_party}(\mathsf{ID})$
  - **else**
    - $\gamma \leftarrow \epsilon$; $\gamma' \leftarrow \epsilon$ *# removed user cleans its states.*
- **if** $T_{op} = (add, \mathsf{ID}, pk) \wedge \mathsf{ID} \neq \mathit{me}$ **then**
  - $\gamma.\texttt{add\_party}(\mathsf{ID}, pk)$
  - *proc-refresh*$(\gamma, T_{new\_keys}, \mathsf{ID}, T_{sender})$ $\gamma.\texttt{update\_pks\_and\_tainter}(T_{new\_pks}, \mathsf{ID}, T_{sender})$
- **if** $T_{op} = welcome$ **then**
  - $\gamma.\texttt{init\_state}(T_{\mathcal{M}}, T_{\mathcal{T}}, T_{\mathcal{H}})$
  - *update-path*$(\gamma, \{\gamma.\texttt{index}(\texttt{me}), \ldots, v_{root}\}, Dec_{sk}(T_{seed}), T_{sender})$
- **if** $T_{op} = \mathit{confirm}$ **then**
  - $\gamma \leftarrow \gamma'$; $\gamma' \leftarrow \epsilon$
- **if** $T_{op} = \mathit{reject}$ **then**
  - $\gamma' \leftarrow \epsilon$
- **if** $T_{op} \notin \{\mathit{confirm}, \mathit{reject}\}$ **then**
  - $\gamma.\texttt{update\_hash}(T)$
- **return**$(\gamma, \mathsf{key}(\gamma))$

2.4 Comparison with Blanking

In terms of security there seems to be little difference between what is achieved using tainting and using blanking. Updates have the same function: they refresh all known secrets, allowing for FS and PCS through essentially the same mechanism in both approaches. We give a security proof for TTKEM below.

However, as mentioned before, tainting seems to be a more natural approach: it maintains the desired tree structure, and its bookkeeping approach gives us a more complete intuition of the security of the tree. It also corresponds to a more flexible framework: as blanking simply forbids parties to know secrets outside of their path the approach leaves little flexibility for how to handle the **init** phase.

With regards to efficiency, the picture is more complicated. TTKEM and TreeKEM are incomparable in the sense that there exist sequences of operations where either one or the other is more efficient. Thus, which one is to be preferred depends on the distribution of operation sequences.

We observe that there are two major differences in how blank and tainted nodes affect efficiency. The first one is which users are affected: a blank node degrades the efficiency of *any user* whose copath contains the blank. Conversely, a tainted node affects only *one* user; the one who tainted it, but on the down side, it does so no matter where in the tree this tainted node is. The second difference is the healing time: to "unblank" a node v it suffices that some user assigned to a leaf in the tree rooted at v performs an update (thereby overwriting the blank with a fresh key). However, to "untaint" v simply overwriting it this way is necessary but not sufficient. In addition, it must also hold that no other node in the tree rooted at v is tainted.

Thus, intuitively, in settings where the tendency is for Adds and Remove operations (i.e. those that produce blanks or taintings) to be (usually) performed by a small subset of group members it is more efficient to use the tainting approach. Indeed, only Update operations done by that subset of users will have a higher cost. As mentioned in the introduction, such a setting can arise quite naturally in practice – e.g. when group membership is managed by a small number of administrators.

In fact, the efficiency benefits of tainting can be further compounded if the users initiating Add/Remove operations also perform Update operations less frequently than others. It turns out that this type of asymmetry between frequency of updates can arise through unrelated (yet realistic) reasons. Suppose, for example, we determine that Bob is at a significantly higher risk of compromise than Alice. Concretely, consider Alice – an admin – who works from the office and Bob, a non-IT professional who communicates using his cellphone in the field. On the one hand, Alice might (be instructed to) only use a well maintained and locked down high security device at the office while accessing the internet through a well defended corporate network. Conversely, Bob's mobile device has a significantly higher risk of falling in the adversaries hands (at least briefly) compared to Alice's device that never leaves the office. Bob might access the internet through a variety of public and private networks. He may also be running a host of other apps on the same device, further raising his risk profile compared to Alice's. Finally, not being a trained IT professional like Alice, he might not be as proficient at preventing compromise; e.g. by detecting fishing attempts, avoiding dubious websites and apps and by using powerful but complicated defensive tools on the device. Under these and similar (quite realistic) types of conditions, it is reasonable to conclude that Bob's device is more likely to be compromised than Alice's, so it is rational to spend a larger proportion of the bandwidth dedicated to a given group chat session on Updates for Bob than the bandwidth spent on Alice's Updates. In particular, this better minimises the probability at any given point that the sessions privacy has been compromised when compared to (say) using the available bandwidth equally between the two.

It should be mentioned that a more thorough analysis could be (and should be) carried out. For example taking into account different probability distributions on the frequency of updates, adds and removes, online/offline behaviour, risk profiles for users to establish what the long term performance of both approaches are under different (realistic) configurations of conditions. We leave this for further work, however.

Efficiency of Initialisation For many group chat sessions, the initialisation phase will be the most inefficient phase of the session's life-cycle. Indeed, inefficiencies arise by adding and removing members to a session. A group will certainly see at least as many Adds as Removes, and likely most of those Adds will happen at the beginning of the group's life (either batched within **init** or just after it). Thus, the process by which a group goes from a initial state to a fully "healed" tree (that without blanks or taints) is of great importance. We will henceforth consider the scenario where a group is initialised with a large number of members and will study the cost (in particular, the number of ciphertexts) needed to transition to a fully healed ratchet tree. We will consider the simplified situation where no further Adds or Removes take place, and where, moreover, the group creator does not perform an update.

While this is obviously quite a restrictive assumption, we believe it would be quite similar to the initial behaviour of most groups. In fact, a similar sequence of group operations could be somehow encouraged by a higher level protocol: a main aim for a group should be to achieve the ratcheting tree structure that gives

log size packages for each operation as soon as possible.[10] We will assume that groups with blanking are initialised as fully blanked trees, except for the creator's path. (To the best of our knowledge, mitigating double-joins via blanking does not allow for any other more efficient initialisation procedure than this.) We also recall that groups with tainting are initialised with a tree fully tainted by the initiator Alice.

In order to fully unblank (resp. untaint) the tree, we need every second member to update. In the tainted case, the order is irrelevant, as any update by a member other than the group creator involves $\log n$ ciphertexts. However, this is not the case with blanking.

Lemma 1. *To transition from a fully blanked (except for the group creator's - the first leaf - path to root) tree to a fully unblanked tree, the following sequence of updates has minimal cost:*

$$n/2+1,\ n/4+1,\ 3n/4+1,\ n/8+1,\ 3n/8+1,\ 5n/8+1,\ 7n/8+1,\ n/16+1,\ \ldots$$

Proof. Let $\mathcal{T}_1$ denote the left subtree, and $\mathcal{T}_2$ the right subtree. If any user (with a leaf) in $\mathcal{T}_1$ updates before anyone in $\mathcal{T}_2$ does, $\mathcal{T}_2$ will be blank and hence one ciphertext per user in $\mathcal{T}_2$ will be needed. On the contrary, if some update from $\mathcal{T}_2$ has already taken place, all updates from $\mathcal{T}_1$ will just need one ciphertext to be communicated to $\mathcal{T}_2$, they will just need to encrypt the new group secret under the head of $\mathcal{T}_2$. Moreover, note that the cost of any update from $\mathcal{T}_2$ will be independent from the structure of $\mathcal{T}_1$, as, being on the group creator's path, the head of $\mathcal{T}_1$ will not be blank. Therefore, the optimal scenario is that someone from the right subtree updates first, assume its the user with the leaf in position $n/2+1$ without loss of generality. Following a similar argument, an update should then come from the right subtree of $\mathcal{T}_1$ before one from its left subtree (similarly for $\mathcal{T}_2$), and so on. □

Now, the cost (i.e. number of required ciphertexts) to update in this order is $(n/2-1)+1$ for the first member, $(n/4-1)+2$ for each of the two next ones, $(n/8-1)+3$ for the 4 next, and so on. We end up with the following lower-bound on the cost of healing:

$$\frac{n}{2}+2\left(\frac{n}{4}+1\right)+4\left(\frac{n}{8}+2\right)+\ldots+\frac{n}{4}\left(\frac{n}{n/2}+\log n-2\right)= \tag{1}$$

$$=\frac{n}{2}(\log n-1)+\sum_{i=1}^{\log n-1}(i-1)2^{i-1} \tag{2}$$

$$=\frac{n}{2}(\log n-1)+2\left(2^{\log n-2}(\log n-2)-2^{\log n-2}+1\right) \tag{3}$$

$$=\frac{n}{2}(2\log n-4)+2 \tag{4}$$

Thus, even for the optimal update ordering, blanking is more costly by about a factor of 2 as the cost for tainting would simply be $(\frac{n}{2}-1)\log n$.

3 Security

3.1 Security Model

Definition 2 (Asynchronous CGKA Security). *The security for CGKA is modelled using a game between a challenger* C *and an adversary* A*. At the beginning of the game, the adversary queries* **create-group**(A, G) *and initialises the group* G *with identities* $(\mathsf{ID}_1,\ldots,\mathsf{ID}_\ell)$*. The adversary* A *can then make a sequence of queries, enumerated below, in any arbitrary order. On a high level,* **add-user** *and* **remove-user** *allows the adversary to control the structure of the group, whereas the queries* **confirm** *and* **process** *allow it to control the scheduling of the messages. The query* **update** *simulates the refreshing of a local state. Finally,* **start_corrupt** *and* **end_corrupt** *enables the adversary to corrupt the users for a time period.*

[10] In particular, the less bandwidth used per Update the more Updates can be performed and so the stronger the expected security properties are for the session will be.

1. **add-user**$(\mathsf{ID}, \mathsf{ID}')$*: a user* ID *requests to add another user* ID' *to the group.*
2. **remove-user**$(\mathsf{ID}, \mathsf{ID}')$*: a user* ID *requests to remove another user* ID' *from the group.*
3. **update**(ID)*: the user* ID *requests to refresh its local state* γ*.*
4. **confirm**(q, β)*: the q-th query in the game, which must be an action* $\mathsf{a} \in \{add, remove, update\}$ *by some user* ID*, is either confirmed (if* $\beta = 1$*) or rejected (if* $\beta = 0$*, in which case* ID *keeps its current state).*
5. **process**(q, ID')*: if the q-th query is as above, this action forwards the (W or T) message to party* ID' *which immediately processes it.*
6. **start_corrupt**(ID)*: from now on the entire view of* ID *is leaked to the adversary.*
7. **end_corrupt**(ID)*: ends the leakage of user* ID*'s internal state* γ *to the adversary.*
8. **challenge**(q^*)*: the adversary picks a query* q^* *which corresponds to an action* $\mathsf{a}^* \in \{add, remove, update\}$ *or the initalization (if* $q^* = 0$*). Let* k_0 *denote the group key that is sampled during this operation and* k_1 *be a fresh random key. The challenger tosses a coin b and – if the safe predicate below is satisfied – the key* k_b *is given to the adversary (if the predicate is not satisfied the adversary gets nothing).*

At the end of the game, the adversary outputs a bit b' *and wins if* $b' = b$*.*
We call a CGKA scheme (Q, ϵ, t)*-CGKA-secure if for any adversary* A *making at most* Q *queries of the form* **add-user**$(.,.)$*,* **remove-user**$(.,.)$*, or* **update**$(.)$ *and running in time* t *it holds*

$$\mathsf{Adv}_{\mathsf{CGKA}}(\mathsf{A}) := |\mathsf{Pr}[1 \leftarrow \mathsf{A}|b = 0] - \mathsf{Pr}[1 \leftarrow \mathsf{A}|b = 1]| < \epsilon.$$

3.2 The safe predicate

We define the *safe predicate* to rule out trivial winning strategies and at the same time restricting the adversary as little as possible. For example, if the adversary challenges the first (create-group) query and then corrupts a user in the group, he can trivially distinguish the real group key from random. Thus, intuitively, we call a query q^* safe if the group key generated in response to query q^* is not computable from any compromised state. Since each group key is encrypted to at most one init key for each party, this means that the users which are group members[11] at time q^* must not be compromised as long as these init keys are part of their state. However, defining a reasonable safe predicate in terms of allowed sequences of actions becomes quite involved.

To gain some intuition, consider the case where query q^* is an update for a party ID^*. Then, clearly, ID^* must not be compromised right after it generated the update. On the other hand, since the update function was introduced to heal a user's state and allow for *post-compromise security* (PCS), any corruption of ID^* *before* q^* should not harm security. Similarly, any corruption of ID^* *after* a further processed update or remove[12] operation for ID^* should not help the adversary either (compare *forward secrecy* (FS)). Finally, also in the case where the update generated at time q^* is rejected to ID^* and ID^* processes this message by returning to its previous state, any corruption of ID^* after processing the reject message should not affect security of the challenge group key. Thus, all these cases should be considered safe.

But we also have to take care of other users which are part of the group when the challenge key is generated: For a challenge to be safe, we must make sure that the challenge group key is not encrypted to any compromised key. At the same time, one has to be aware of the fact that in the asynchronous setting the view of different users might differ substantially. We consider inconsistency of user's states rather a matter of functionality than security, and aim to define the safe predicate as unrestrictive as possible, to also guarantee security for inconsistent group states. For example, consider the following scenario: A user ID generates an update during an uncompromised time period and processes a reject for this update still in the uncompromised time period, but this update is confirmed to and processed by user ID^* before he does his challenge update q^*; then this results in a safe challenge since the challenge group key is only encrypted to the new init key which is not part of ID's state at any compromised time point. However, one has to be

[11] To be precise, since parties might be in inconsistent states, group membership is not unique but rather depends on the users' *views* on the group state. We will discuss this below.

[12] Note, when processing a remove operation, the target party deletes its current state and draws a fresh init key, which can later be used if the party was added to the group again.

careful here, since in a similar scenario where ID receives a reject for his update but does not process it, the challenge group key would clearly not be safe anymore.

In the following definition, we capture all different safe sequences of actions for all possible challenge queries. To avoid introducing a sign-up algorithm, we assume all parties are always signed-up, i.e., they always have an init key pair which is used whenever they are added, can be compromised by the adversary even before the party was added, can be updated any time, and are reset to a fresh init key pair with each remove. We consider discrete time steps measured in terms of the number of queries that have been issued by the adversary so far.

We begin with two definitions that allow us to capture exactly what we mean by PCS and FS. First we define at which point a user is considered not compromised (anymore).

Definition 3 (Not Compromised). *For two users* ID *and* ID^**, we say that user* ID *is* not compromised *at time* q^* *in* ID^**'s view, if the last message* a_{ID} *of the form* $\mathbf{update}(\mathsf{ID}), \mathbf{remove\text{-}user}(\cdot, \mathsf{ID}), \mathbf{create\text{-}group}(\cdot, G)$ *processed by* ID^* *before* q^* *was generated at a time where* ID *was not corrupted.*

We now define when a user cannot leak past secrets anymore, i.e. at which point FS holds. We note that the second to fourth points in the following definition are technical corner cases, which are included for completeness. On first read, we recommend focusing on the first point.

Definition 4 (Moved On). *For users* ID *and* ID^* *and time* q^**, let* a_{ID} *be the last message of the form* $\mathbf{update}(\mathsf{ID}), \mathbf{remove\text{-}user}(\cdot, \mathsf{ID}), \mathbf{create\text{-}group}(\cdot, G)$ *processed by* ID^* *before* q^**. We say that user* ID *has* moved on *from* q^* *relative to* ID^**, if*

- ID *processed some* $\mathsf{a}'_{\mathsf{ID}} \in \{\mathbf{update}(\mathsf{ID}), \mathbf{remove\text{-}user}(\cdot, \mathsf{ID})\}$ *before the next corruption of* ID *and* $\mathsf{a}'_{\mathsf{ID}}$ *was not processed by* ID^* *before* q^**, or*
- ID *processed a reject of* a_{ID} *(only possible when* $\mathsf{a}_{\mathsf{ID}} = \mathbf{update}(\mathsf{ID})$*) before the next corruption of* ID*, or*
- $\mathsf{a}_{\mathsf{ID}} = \mathbf{remove\text{-}user}(\cdot, \mathsf{ID})$ *and* ID *never processed* a_{ID} *or any later remove or update, i.e.,* ID *never switches to the next init key, or*
- ID *is never corrupted after the generation of* a_{ID}*.*

We are now ready to define when a key should be considered *safe*.

Definition 5 (Safe predicate). *Let* sk^* *be a group key generated on behalf of user* ID^* *in an action* $\mathsf{a}^* \in \{\mathbf{add\text{-}user}(\mathsf{ID}^*_{\mathsf{gen}}, \mathsf{ID}^*), \mathbf{remove\text{-}user}(\mathsf{ID}^*_{\mathsf{gen}}, \mathsf{ID}^*), \mathbf{update}(\mathsf{ID}^*)\}$ *at time point* q^* *and let* G^* *be the set of users which would end up in the group if query* q^* *was processed, as viewed by the generating user* $\mathsf{ID}^*_{\mathsf{gen}}$*, where* $\mathsf{ID}^*_{\mathsf{gen}} := \mathsf{ID}^*$ *for* $\mathsf{a}^* = \mathbf{update}(\mathsf{ID}^*)$*. Then the key* sk^* *is called* safe *if the following are true:*

- *For all* $\mathsf{ID} \in G^* \setminus \{\mathsf{ID}^*\}$ *we require that* ID *is* not compromised *at time* q^* *in* $\mathsf{ID}^*_{\mathsf{gen}}$*'s view and* has moved on *from time* q^* *relative to* $\mathsf{ID}^*_{\mathsf{gen}}$*.*
- *For* ID^* *we require that*
  - *if* $\mathsf{a}^* = \mathbf{add\text{-}user}(\mathsf{ID}^*_{\mathsf{gen}}, \mathsf{ID}^*)$*: that* ID^* *is* not compromised *at time* q^* *in* $\mathsf{ID}^*_{\mathsf{gen}}$*'s view and* has moved on *from* q^* *relative to* $\mathsf{ID}^*_{\mathsf{gen}}$*.*
  - *if* $\mathsf{a}^* = \mathbf{update}(\mathsf{ID}^*)$*: that* ID^* *is not corrupted at time* q^* *and* has moved on *from* q^* *relative to itself.*

If $\mathsf{a}^* = \mathbf{create\text{-}group}(\mathsf{ID}^*_{\mathsf{gen}}, G^*)$*, then* sk^* *is safe if all* $\mathsf{ID} \in G^*$ *have moved on from time* $q^* = 0$ *relative to* $\mathsf{ID}^*_{\mathsf{gen}}$*.*

A challenge query q^* *is called* safe *if the secret key generated at time* q^* *is safe.*

3.3 Security Proof for TTKEM

To prove security of TTKEM, we reduce the security game to a variant of a game called *generalized selective decryption* (GSD), which was introduced by Panjwani [12] in order to prove adaptive security of multicast encryption protocols. While GSD was originally defined in the symmetric-key setting, adaptive security in the public-key setting can be proven in a similar way, as discussed below. In fact, we will consider a more restricted variant of the game and use the framework of Jafargholi et al. [10] to get better security guarantees than in the original case.

The original GSD game (in the symmetric-key setting) is defined as follows:

Definition 6 (Generalized selective decryption (GSD), [12]). *Let* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *be a symmetric-key encryption scheme with key space* $\mathcal{K}$ *and message space* $\mathcal{M}$ *such that* $\mathcal{K} \subseteq \mathcal{M}$*. The* GSD game *is a two-party game between a challenger* C *and an adversary* A*. On input integers* N, Q *and the security parameter* λ*, for each* $v \in [N]$ *the challenger* C *picks a key* $k_v \leftarrow \mathsf{KeyGen}(1^\lambda)$ *and initializes the* key graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}) := ([N], \emptyset)$ *and the set of corrupt users* $\mathcal{C} = \emptyset$*.* A *can adaptively do the following queries:*

- $(\mathit{encrypt}, u, v)$*: On input two nodes* u *and* v*,* C *returns an encryption* $\mathsf{Enc}_{k_u}(k_v)$ *of* k_v *under* k_u *and adds the directed edge* (u, v) *to* $\mathcal{E}$*.*
- $(\mathit{corrupt}, v)$*: On input a node* v*,* C *returns* k_v *and adds* v *to* $\mathcal{C}$*.*
- $(\mathit{challenge}, v)$*, single access: On input a challenge node* v*,* C *samples* $b \leftarrow \{0, 1\}$ *uniformly at random and returns* k_v *if* $b = 0$*, otherwise* $k \leftarrow \mathcal{K}$ *uniformly at random. The challenge node* v *must be a sink and must not be reachable (by following the directed edges) from any node in* $\mathcal{C}$ *as otherwise determining* b *is trivial. We also require that* $\mathcal{G}$ *is acyclic (for reasons discussed below).*

Finally, A *outputs a bit* b' *and it* wins *the game if* $b' = b$*. We call the encryption scheme* (Q, ϵ, t)*-adaptive GSD-secure if for any adversary* A *making at most* Q *queries and running in time* t *it holds*

$$\mathsf{Adv}_{\mathsf{GSD}}(\mathsf{A}) := |\mathsf{Pr}[1 \leftarrow \mathsf{A} | b = 0] - \mathsf{Pr}[1 \leftarrow \mathsf{A} | b = 1]| < \epsilon.$$

Reducing the security of the (adaptive) GSD game to the IND-CPA security of the underlying encryption scheme is in general not an easy task. One must require that $\mathcal{G}$ is acyclic as otherwise a reduction is impossible because IND-CPA does not imply any kind of circular security.

Even with this restriction, the best known reductions involve an exponential (in the size of the graph) loss in security. However, for versions of GSD where the adversary's query behaviour is restricted to certain graph structures like trees or graps of low depth, much better reductions are known (see [12], [10]).

We are not aware of any previous work on variants of GSD in the *public-key* setting and one needs to be careful here since a naive adaptation of the original GSD game (where the adversary knows all public-keys, and only the secret-keys get encrypted) would be trivial to win: If the adversary knows the public key of the secret key encrypted in the challenge node, then it can obviously distinguish this secret key from a uniformly random key. We consider a public-key variant of GSD which will capture TTKEM while not being trivial. Concretely, instead of assigning a key pair to each node, we consider the scenario where a subset of nodes is instead assigned random strings and the adversary is restricted to only query a challenge from this distinguished subset. This imposes a restriction to the encryption queries since these potential challenge nodes naturally can only be target nodes of an encryption, hence, sinks in the key graph. Security for this public-key GSD game follows from known results on GSD in the symmetric-key setting.

First, we will analyse a slightly different version of TTKEM, which is directly related to the GSD game, where, for each initialize/add/remove/update operation, instead of generating key paths from a secret seed using the hierarchical key derivation mechanism, we assume the new keys along the path were indepedent and can be derived only by querying an oracle. We call this scheme TTKEM$^-$ and consider the following restricted GSD game in the public-key setting (see Figure 6).

Definition 7 (Restricted generalized selective decryption (GSD), public-key setting). *Similar to Definition 6, let* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *be a public-key encryption scheme, with secret key space* $\mathcal{K}$ *and message space* $\mathcal{M}$ *for the public-key scheme such that* $\mathcal{K} \subset \mathcal{M}$*. Furthermore, let* $\mathcal{S}$ *be a set of seeds with* $\mathcal{S} \subset \mathcal{M}$*,*

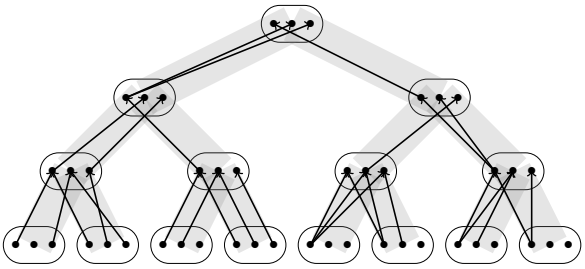


Fig. 6. Possible key graph for restricted GSD with $Q = 2$ and $n = 8$. The underlying binary tree structure is represented shaded grey.

and let $\mathcal{T} = ([2n-1], \mathcal{E}_\mathcal{T})$ *be a directed binary tree with sink/root* $j_{\mathsf{root}} = 1$ *and sources/leaves* $[n, 2n-1]$*, i.e., labeled from root to leaves from left to right. In the* restricted GSD *game, on input integers* n, Q *and the security parameter* λ*, the challenger* C *picks a key pair* $(\mathsf{pk}_{i,j}, \mathsf{sk}_{i,j}) \leftarrow \mathsf{KeyGen}(1^\lambda)$ *for each* $(i,j) \in [Q+1] \times [2, 2n-1]$*, a uniformly random seed* $\mathsf{sk}_{i,1} \leftarrow \mathcal{S}$ *for each* $i \in [Q+1]$*, and initializes the* key graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}) := ([Q+1] \times [2n-1], \emptyset)$ *and the set of corrupt users* $\mathcal{C} = \emptyset$*. Then* C *sends all the public keys* $\mathsf{pk}_{i,j}$ *to* A*.* A *can adaptively do the following queries:*

- $(\mathit{encrypt}, (i,j), (i',j'))$*: On input two nodes* (i,j) *and* (i',j')*,* C *returns an encryption* $\mathsf{Enc}_{\mathsf{pk}_{i,j}}(\mathsf{sk}_{i',j'})$ *of* $\mathsf{sk}_{i',j'}$ *under* $\mathsf{pk}_{i,j}$ *and adds* $((i,j),(i',j'))$ *to* $\mathcal{E}$*. However,* A*'s queries are restricted such that*
  - $(j,j') \in \mathcal{E}_\mathcal{T}$*,*
  - *for all* $(i,j) \in \mathcal{V}$*:* $\mathsf{indeg}((i,j)) \leq 2$ *and* $\mathsf{indeg}((i,j)) = 2$ *is only allowed if* $j \in [n/2, n-1]$*;*
- $(\mathit{corrupt}, (i,j))$*: On input a node* (i,j)*,* C *returns* $sk_{i,j}$ *and adds* (i,j) *to* $\mathcal{C}$*.*
- $(\mathit{challenge}, (i,j))$*, single access: On input a challenge node* (i,j)*,* C *samples* $b \leftarrow \{0,1\}$ *uniformly at random and returns* $\mathsf{sk}_{i,j}$ *if* $b = 0$*, otherwise a freshly sampled uniformly random seed* $\mathsf{sk} \leftarrow \mathcal{S}$*. However, it must hold*
  - j *is the root of* $\mathcal{T}$*, i.e.,* $j = 1$*,*
  - (i,j) *is not reachable in* $\mathcal{G}$ *from any node in* $\mathcal{C}$ *(this must hold throughout the entire game).*

Finally, A *outputs a bit* b' *and it* wins *the game if* $b' = b$*. We call the encryption scheme* (Q, ϵ, t)*-adaptive restricted GSD-secure if for any adversary* A *making at most* Q *queries and running in time* t *it holds*

$$|\mathsf{Pr}[1 \leftarrow \mathsf{A} | b = 0] - \mathsf{Pr}[1 \leftarrow \mathsf{A} | b = 1]| < \epsilon.$$

While in the CGKA security game from Definition 2 the adversary is only allowed to corrupt users at their current state, for simplicity we defined a stronger version of GSD where the adversary is allowed to corrupt arbitrary nodes in the key graph and, in particular, can also corrupt a user at any previous state.

The family of graphs the adversary is allowed to query in the restricted variant of GSD as defined in Definition 7 clearly captures (supergraphs of) the graphs which are constructed during an execution of Game 2 on TTKEM$^-$ with maximal group size n, maximal Q add/remove/update queries, and fixed depth $\log n$ (see Figure 7): When initializing the group, we obtain a subgraph of the complete binary tree on $2n-1$ nodes (shaded in Figure 7) consisting of disjoint "forked" paths of maximal length $\log n$, where we call a graph a forked path if it consists of a path and at most one additional edge incident on the second node in the path. Each add/remove/update query refreshes some of the nodes in that tree and adds the corresponding edges incident on these new nodes; note that – except for children of leaves – each fresh key is encrypted to one parent, hence, only one edge incident on each new node is added. Hence, after a total of Q add, remove, and update queries of the adversary, each of these $2n-1$ nodes was refreshed at most Q times and for each of the $Q+1$ group keys the set of ancestors forms a forked path within the complete binary tree on $2n-1$ nodes.

To reduce the security of TTKEM$^-$ to GSD, it remains to prove that a *safe* challenge group key in the game of Definition 2 corresponds to a valid GSD challenge.

Lemma 2. *Any safe group key is a valid challenge in the restricted GSD game.*

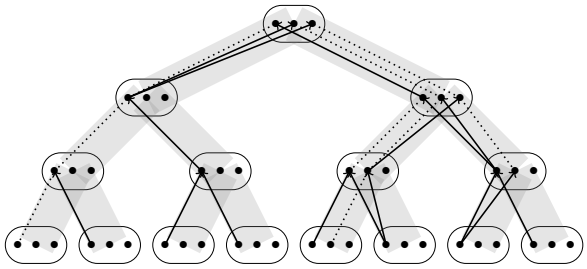


Fig. 7. Graph structure generated by A when initializing a group of maximal supported size $n = 8$ on behalf of the first user and doing $Q = 2$ queries, first an update query for the fifth identity, then a remove query targeting the eighth identity. The dotted edges represent keys created by hierarchical key derivation, solid edges represent encryptions.

Proof. Let $(\mathsf{pk}^*, \mathsf{sk}^*) = (\mathsf{pk}_{i,j}, \mathsf{sk}_{i,j})$ be a safe group key, corresponding to node (i, j) in $\mathcal{G}$, which was generated in response to query q^*. Since (i, j) is a group key, it must hold that j is the sink of the binary tree $\mathcal{T}$ and, thus, (i, j) is a sink of $\mathcal{G}$. Furthermore, by the structure of the encryption queries induced by the init/add/remove/update queries in the CGKA security game, the ancestor graph of the challenge node (i, j) must be a "forked" path. To prove the claim, we must show that none of the nodes in this forked path is compromised.

As in Definition 5, let the q^*th query be an action $\mathsf{a}^* \in \{\textbf{create-group}(\mathsf{ID}^*_{\mathsf{gen}}, G^*), \textbf{add-user}(\mathsf{ID}^*_{\mathsf{gen}}, \mathsf{ID}^*), \textbf{remove-user}(\mathsf{ID}^*_{\mathsf{gen}}, \mathsf{ID}^*), \textbf{update}(\mathsf{ID}^*)\}$ and let G^* be the set of group members at state q^* as viewed by $\mathsf{ID}^*_{\mathsf{gen}}$. Since q^* is safe, either, before generating a^*, party $\mathsf{ID}^*_{\mathsf{gen}}$ must have processed an action $\mathsf{a}_{\mathsf{ID}} \in \{\textbf{create-group}(., G), \textbf{remove-user}(., \mathsf{ID}), \textbf{update}(\mathsf{ID})\}$ (with $\mathsf{ID} \in G$) for each party $\mathsf{ID} \in G^*$, or $\mathsf{a}^* = \textbf{create-group}(\mathsf{ID}^*_{\mathsf{gen}}, G^*)$ and $\mathsf{a}_{\mathsf{ID}} := \mathsf{a}^*$ for all $\mathsf{ID} \in G^*$, and it must hold that all parties $\mathsf{ID} \in G^*$ are not compromised during the generation of a_{ID} and

- ID was either never corrupted after the generation of a_{ID}, or
- ID never switches its init key to the one considered by $\mathsf{ID}^*_{\mathsf{gen}}$ when generating a^*, or
- ID was not corrupted during the time period between a_{ID}'s generation and the time when ID's entire state is refreshed by either processing a $\mathsf{reject}(\mathsf{a}_{\mathsf{ID}})$ or processing an action $\mathsf{a}'_{\mathsf{ID}} \in \{\textbf{remove-user}(., \mathsf{ID}), \textbf{update}(\mathsf{ID})\}$ which is not confirmed to $\mathsf{ID}^*_{\mathsf{gen}}$ before q^*.

In the case $\mathsf{a}^* = \{\textbf{create-group}(\mathsf{ID}^*_{\mathsf{gen}}, G^*)$, this immediately implies that none of the nodes in the ancestor graph can be corrupted. In the other cases, since $\mathsf{ID}^*_{\mathsf{gen}}$ only processes actions with consistent history, there must be an ordering of the actions $\{\mathsf{a}_{\mathsf{ID}}\}_{\mathsf{ID} \in G^*} \cup \{\mathsf{a}^*\}$ as $\mathsf{a}_1, \ldots, \mathsf{a}_{\ell-1}, \mathsf{a}_\ell := \mathsf{a}^*$ with a_i generated by party $\mathsf{ID}^i_{\mathsf{gen}}$ during query q_i such that, for all $i \in [\ell]$, party $\mathsf{ID}^{i+1}_{\mathsf{gen}}$ processed a_i before generating a_{i+1}. In particular, in the ancestor graph of the sink node generated in query q_{i+1}, all the nodes which are only known to the users $\mathsf{ID}^1_{\mathsf{gen}}, \ldots, \mathsf{ID}^i_{\mathsf{gen}}$ are not compromised. (This easily follows by induction.) This implies that none of the nodes in the ancestor graph of the challenge node is compromised, which proves the claim. □

Note, for the proof of Lemma 2 it is crucial that updates, removes, and adds not only refresh the keys along the path from a party's leaf to the sink in the tree, but also refresh all keys labelled by this party.

It remains to reduce the security of our restricted version of GSD to the IND-CPA security of the encryption scheme. Naively using the result from [12] for GSD restricted to graphs of bounded depth gives the following (note, in our case the key graph is a DAG of depth $\log n$):

Theorem 1 ([12]). *If the encryption scheme* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *is* (ϵ, t)*-IND-CPA secure, then it is also* $(Q, \epsilon \cdot O((2(Q+1)(2n-1))^{\log n+1}), t)$*-adaptive restricted GSD secure.*

However, if we apply the framework of Jafargholi et al. [10] and take the known structure of the graph into account, we get a much stronger security bound:

Theorem 2. *If the encryption scheme* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *is* (ϵ, t)*-IND-CPA secure, then it is also* $(Q, \epsilon \cdot \frac{n}{2} \cdot (3(Q+1))^{\log\log n+1}, t)$*-adaptive restricted GSD secure.*

Proof. Following the framework of Jafargholi et al. [10], to reduce restricted GSD to IND-CPA, one can define a sequence of ϵ-indistinguishable hybrid games from reversible edge pebbling as defined in [10]: If there exists a valid edge pebbling sequence on the key graph, starting with the empty configuration and ending with the configuration where only the two edges incident on the challenge node are pebbled, and this sequence has length ℓ and each configuration can be represented with s bits, then the encryption scheme is $\epsilon \cdot \ell \cdot 2^s$-adaptive GSD secure. In the restricted GSD game, for any safe CGKA challenge it holds that the corresponding node in $\mathcal{G}$ is a sink node and the set of ancestors forms a path of length $\log n$ with at most one additional edge incident on the second node on the path. To pebble the challenge graph, one basically needs to pebble the path. However, for paths it is a known result [5] that there exists a reversible pebbling strategy of length $\ell = 3^{\log\log n+1}$ which uses at most $\log\log n + 1$ pebbles at each step. This implies a reversible pebbling of the challenge graph of length at most $2 \cdot \ell$.
By the structure of the key graph $\mathcal{G}$ we have the following: First, there exists a $k \in [n/2]$ such that for each node (i, j) on the challenge path it holds either $j = 2k + 1$ or j lies on the unique path from $2k$ to 1 in the binary tree $\mathcal{T}$. Second, for each $j \in [2n - 1]$ there is at most one $i \in [Q + 1]$ such that (i, j) is in the challenge graph. Thus, for each pebbling configuration on the challenge path the set of pebbled edges can be represented using at most $s = (\log n - 1) + (\log\log n + 1)\log(Q + 1)$ bits. This implies adaptive restricted GSD-security at a loss in security $\frac{n}{2} \cdot (3(Q + 1))^{\log\log n+1}$. □

Using the above results, we obtain security for TTKEM.

Corollary 1. *If the underlying encryption scheme is (ϵ, t)-IND-CPA secure, then TTKEM with maximal n users is $(Q, \epsilon \cdot \frac{n}{2} \cdot (3(Q + 1))^{\log\log n+1}, t)$-CGKA-secure in the random oracle model.*

Proof. For TTKEM$^-$, the claim follows directly from Theorem 2. For ease of analysis, we assume that all users always retain the seed for the key generation algorithm along with the keys in their state in TreeKEM$^-$.[13] Note that this does not impact the security of TTKEM$^{(-)}$, since one can simply view the seed as part of the secret key. Any encryption scheme that is IND-CPA secure remains IND-CPA secure with this change. Thus, if we call this intermediate scheme TTKEM', it must hold that TTKEM' is *at least as secure* as TTKEM$^-$. It remains to prove that the original construction TTKEM is as secure as TTKEM'. However, this easily follows in the random oracle model: Assume there was an adversary A breaking the CGKA security of TTKEM, where the hash functions H_2 is modelled as a random oracle, ie., A only has oracle access to this uniformly distributed function. Then we can construct a reduction R which uses A as a black box and breaks the CGKA security of TTKEM' with similar advantage as follows: Except for the corrupt queries, R simply forwards all queries from A to its own oracles and returns whatever it receives. Since the output of H_2 is uniformly random, the distribution of keys associated to the nodes in the tree is exactly the same when using the hierarchical key derivation mechanism as when sampling the keys independently. However, when A corrupts a party i it learns a set of keys and expects them to follow the correct distribution, i.e. they should all be derived from the same seed. Now recall that the users in TreeKEM' retain the seeds for the key generation algorithm so when R corrupts a party (which it does whenever A does) it learns the seed Δ that was used to generate the keys. So the reduction can sample a uniformly random seed, derive the path of seeds by iteratively applying H_1 and then *program* H_2 to output Δ on input Δ_i. This proves the claim.[14] □

Remark 1. For blanked TreeKEM a similar approach can be used to prove adaptive security in the random oracle model. However, in this case one needs to consider GSD restricted to a different graph structure where the ancestor graph of any possible challenge sink/root consists of a path of maximal length $\log n$ starting at one of n possible buckets of leaves (corresponing to the members of the group), and up to n additional edges from distinct leaves to nodes on that path. To reversibly pebble that graph, one basically needs to pebble

[13] In many cryptosystems the seed is computable from the secret key so this is a fairly natural way of viewing the state.

[14] We do not go into details here and neglect the loss in security involved by collisions, which will anyway be dominated by the loss given by the security reduction for GSD.

the path, where the time to pebble/unpebble a node takes up to n additional steps, and each node on the path can be represented using $\log(Q+1)$ bits. Thus, one needs at most $s = \log n + (\log\log n + 1)\log(Q+1)$ bits to represent each of the $\ell < n \cdot 3^{\log\log n+1}$ pebbling configurations. This implies adaptive security of TreeKEM with blanking at a security loss $< n^2 \cdot (3(Q+1))^{\log\log n+1}$ in the random oracle model.

Remark 2. Considering encryption schemes which satisfy the stronger CCA security, we can define the corresponding GSD game by additionally giving the adversary access to an encryption as well as a decryption oracle. Security of GSD in the CCA setting can then be reduced to the CCA security of the underlying encryption scheme at the same loss in security. In particular, an analogous version of Corollary 1 holds for CCA-secure schemes.

References

1. Message Layer Security (mls) WG. https://datatracker.ietf.org/wg/mls/about/.
2. J. Alwen, S. Coretti, and Y. Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. In Y. Ishai and V. Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 129–158. Springer, Heidelberg, May 2019.
3. J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis. Security analysis and improvements for the ietf mls standard for group messaging. Cryptology ePrint Archive, Report 2019/1189, 2019. https://eprint.iacr.org/2019/1189.
4. Bhargavan, Karthikeyan and Barnes, Richard and Rescorla, Eric. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups. May 2018.
5. F. Chung, P. Diaconis, and R. Graham. Combinatorics for the East Model. Advances in Applied Mathematics 27, pages 192-206, 2001.
6. K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, and K. Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 1802–1819. ACM Press, Oct. 2018.
7. F. B. Durak and S. Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In N. Attrapadung and T. Yagi, editors, *Advances in Information and Computer Security*, pages 343–362, Cham, 2019. Springer International Publishing.
8. G. Fuchsbauer, Z. Jafargholi, and K. Pietrzak. A quasipolynomial reduction for generalized selective decryption on trees. In R. Gennaro and M. J. B. Robshaw, editors, *Advances in Cryptology – CRYPTO 2015, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 601–620. Springer, Heidelberg, Aug. 2015.
9. IETF. The Messaging Layer Security (MLS) Protocol (draft-ietf-mls-protocol-07). https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/, July 2019.
10. Z. Jafargholi, C. Kamath, K. Klein, I. Komargodski, K. Pietrzak, and D. Wichs. Be adaptive, avoid overcommitting. In J. Katz and H. Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 133–163. Springer, Heidelberg, Aug. 2017.
11. D. Jost, U. Maurer, and M. Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Y. Ishai and V. Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 159–188. Springer, Heidelberg, May 2019.
12. S. Panjwani. Tackling adaptive corruptions in multicast encryption protocols. In S. P. Vadhan, editor, *TCC 2007: 4th Theory of Cryptography Conference*, volume 4392 of *Lecture Notes in Computer Science*, pages 21–40. Springer, Heidelberg, Feb. 2007.