Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement

Joël Alwen[1], Margarita Capretto[2], Miguel Cueto[3], Chethan Kamath[4], Karen Klein[5⋆], Ilia Markov[5], Guillermo Pascual-Perez[5⋆⋆], Krzysztof Pietrzak[5⋆], Michael Walter[5⋆], and Michelle Yeo[5⋆⋆]

[1] Wickr Inc.
[2] Universidad Nacional de Rosario
[3] ENS de Lyon
[4] Northeastern University
[5] IST Austria

Abstract. While messaging systems with strong security guarantees are widely used in practice, designing a protocol that scales efficiently to large groups and enjoys similar security guarantees remains largely open. The two existing proposals 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).
The first contribution of this paper is a new protocol, that we call Tainted TreeKEM (TTKEM for short), which we show through simulations to be more efficient than TreeKEM for some natural distributions of add and remove operations.
Our second contribution are security proofs for TTKEM and TreeKEM which show that these protocols provide post compromise security and forward security even against active and adaptive attackers. Concretely, in the standard model we bound the security loss (to the underlying PKE) by a quasipolynomial factor $Q^{\log(n)}$, where n is the group size and Q the total number of (update/remove/invite) operations. This proof is in the framework of Jafargholi et al. [Crypto'17]. In the random oracle model we prove a polynomial $(Qn)^2$ bound for TTKEM; the same bound can be obtained for TreeKEM.

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 achieve 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].

⋆ 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.

Instead of constructing a messaging scheme directly, a modular approach seems more natural. This was done for the two party case by Alwen *et al.* [2]. We consider in this paper the concept of Continuous Group Key Agreement Protocol CGKA, a generalisation for groups of their Continuous Key Agreement (CKA). Such a primitive can then be used to build a group messaging protocol in a fashion similar to the one in their paper.

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[6] 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 one.

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) by Cohn-Gordon *et al.* [8]. This protocol (as well as TreeKEM, discussed below, and our protocol) identifies the group with a binary tree where edges are directed and point from the leaves to the root.[7] Each party ID_i in the group is assigned their own leaf, which 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 parents have secret values a and b has the secret value g^{ab} and public value $g^{\iota(g^{ab})}$, where ι is a map to the integers. The secret value of the root is the group key. As illustrated in Figure 1, 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 efficiently add or remove parties in the ART protocol, or even initialise a group, without at least some of the parties (apart from the initiator of the operation) being online.

The authors prove the ART protocol secure even against adaptive adversaries. However, 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 that is super-linear in the group size, resulting in large messages and defeating the whole purpose of using a tree structure.

[6] We use a capital letter (Update, Add, Remove) to refer to the operation (not the verb).

[7] 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.

TreeKEM. The TreeKEM proposal [6, 15] 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 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, she simply samples a completely fresh path from a (fresh) leaf to a (fresh) root replacing the path from ID_j's leaf to the root. She 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. ID_i can add ID_j similarly, 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 knows 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 said to be "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[8]. 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.

Let us mention that since the November 2019 version of the MLS draft, the design of TreeKEM is slightly more involved. Operations are bundled and executed in groups, in what are called Commits. A user can decide to just propose an operation, which does not change the state of the group, or commit and move

[8] The Add operation was modified in the 9th version of the MLS draft (march 2020) and it no longer creates blanks. We omit the details for simplicity and for being somewhat orthogonal to the discussion, and refer the interested reader to [15]. We remark, however, that the efficiency comparison from Section 2.4 considers the latter, more efficient version of TreeKEM

the group forward into a new epoch. A Commit entails executing all proposed operations since the last Commit, together with an Update of the user committing. TTKEM does not use Commits, and hence we will omit the exact details of the construction.

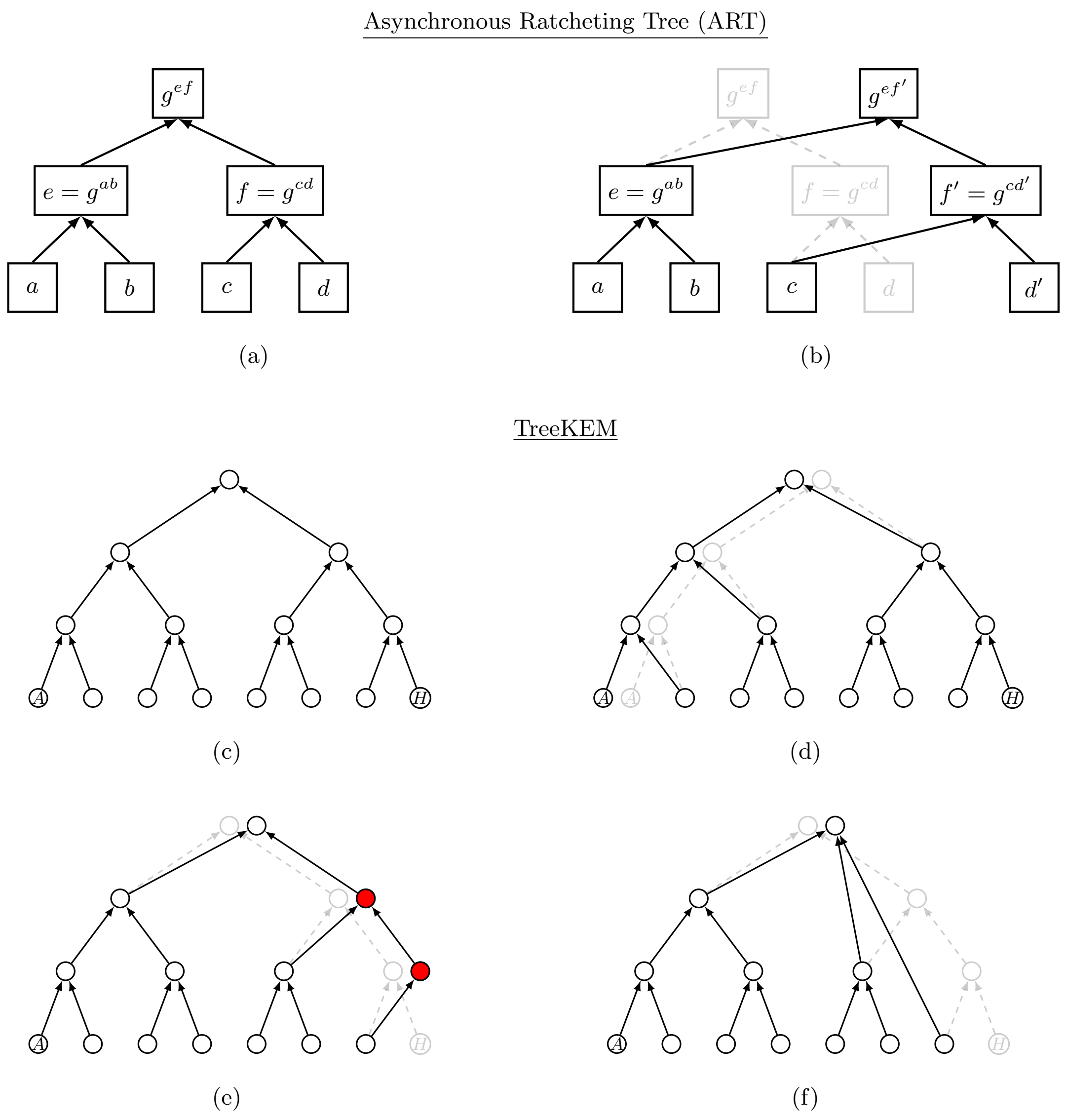


Fig. 1: **Top**: Illustration of an Update in the ART protocol. The state of the tree changes from (a) to (b) when Dave (node d) updates his internal state to d'. **Bottom**: Update and Remove in TreeKEM and TreeKEM with blanking. The state of a completely filled tree is shown in (c). The state changes 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.

1.2 Our Contribution

In this work we propose a new protocol, called Tainted TreeKEM, or simply TTKEM, which is similar to TreeKEM but more efficient on certain realistic scenarios. 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.

It was brought to our attention that a post on the IETF mailing list[9] 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 definition of the security model and the security 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 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 an invariant which states that the secret key of any (non-blanked) node 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 knows their value, but it is instructive to analyze 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 anyway. 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; something an Update should have prevented (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 do not 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 refer to nodes whose secret keys were created by a party ID_i which is not an 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 cases where ID_i performs an Update or is removed, in which 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. The intuition for this is that 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.* In Section 2.4 we show that for some natural cases TTKEM will significantly outperform TreeKEM. This improvement is most patent in the case where a small subset of parties perform most of the Add and Remove operations. 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.

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

[9] [MLS] Removing members from groups Jon Millican {jmillican@fb.com} 12 February 2018 `https://mailarchive.ietf.org/arch/msg/mls/4-gvXpc-LGbWoUS7DKGYG65lkxs`

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 do not differ in this aspect and efficiency-wise there is not much left to improve.[10]

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 FS and PCS, 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 FS and PCS.

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 the 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 relay a message. A more detailed description is given below.

The adversary can choose to corrupt any party, after which its state becomes fully visible to the adversary. In particular, corrupting a party gives the adversary access to the random coins used by said party when executing any group operation, deeming the party's actions deterministic in the eyes of the adversary throughout the corruption. We would like to stress that security in this strong model implies security in weaker and potentially more realistic models, e.g. consider the setting where malware in a device leaks some of the randomness bits but cannot modify them. 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

[10] 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.

(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, does not enforce an ordering), then we cannot hope for liveness, but we still achieve meaningful 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 [10, 18].[11]

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 reflects the fact that the use of signatures would prevent a compromised server from creating ciphertexts that would then be accepted by group members. A group member could indeed adversarially create malformed messages and break correctness, but we do not see an attack on privacy based on this. Extending the proof to such a fully active security model is a major open problem in the area and a great one.

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 in the standard model via piecewise guessing. We reduce the security of the protocol to the security of the underlying encryption scheme. By using the framework of Jafargholi et al. [17] (often refered to as *piecewise guessing*) we achieve a quasipolynomial security loss (in the order of $Q^{\log(n)}$, 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.

We observe that the security of TTKEM resembles the generalised selective decryption (GSD) game [20]. In this multi-user game in the *secret-key* setting, an adversary can ask for encryptions of secret keys under other secret keys, e.g. a query (i, j) would return the ciphertext $\mathsf{Enc}_{sk_i}(sk_j)$ of sk_j under pk_i. We think of the sk_i's as nodes in a graph, and add a directed edge $i \to j$ when the adversary makes a query $\mathsf{Enc}_{sk_i}(sk_j)$. The adversary can also make corruption queries, where on input i he gets sk_i. The adversary can then challenge an index i, but to be a valid challenge it must hold that (1) sk_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 $\mathsf{Enc}_{sk_i}(m_0)$ from $\mathsf{Enc}_{sk_i}(m_1)$ for m_0, m_1 of its choice).

The TTKEM (and also TreeKEM) security experiment can also be viewed as a game where the adversary adaptively queries edges of an acyclic graph. In contrast to GSD, here there are *two* types of edges - the hash function evaluations in the hierarchical derivation of seeds and encryption edges. (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 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 query responses or even getting the public keys. Proving security for an adaptive adversary is much more challenging, a standard complexity leveraging argument would lose an exponential (in $|V| \approx n \cdot q$) factor. For the GSD game, the works of Panjwani [20] and Fuchsbauer *et al.*[12] (which both fall into the framework of [17]) improve this to quasipolynomial in the depth of the graph. A generalisation of their results to our setting would give us $\approx (Q \cdot n)^{2\log(n)} \cdot \epsilon$

[11] 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.

security, which is already much better than an exponential loss. This bound only relies on the size and depth of the underlying graph. Taking the more restrictive structure of the queries and the graph constructed in the TTKEM security game into account we can improve this to $\approx Q^{\log(n)} \cdot \epsilon$. In a nutshell, we save an n factor in the base because with every encryption query, we exactly know at which depth in the tree it will end up, so we do not have to guess that. We save a factor 2 in the exponent because a query in TTKEM implies a very structured sequence of encryption queries in the underlying game.

The reduction in the ROM. In (Tainted) TreeKEM, a node is identified with a short seed s, from which the public/secret key pairs of this node are derived. If the randomness used to sample those keys is a hash of s, and we model this hash as a random oracle, we can give a much better *polynomial* bound for the adaptive security of both TTKEM and TreeKEM.

This proof is very different from the proof in the standard model and employs a new result on a *public-key* version of GSD in the random oracle model. To the best of our knowledge this is the first time GSD is defined and analysed in the public-key setting and we believe this result to be of independent interest. The bound we get is *independent* of the graph structure underlying the GSD game, hence can easily be adapted to TreeKEM with blanking as well.

The safe predicate. We still need to understand what the security in the GSD-like 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.

Our approach of reducing the security of TTKEM (and TreeKEM) to GSD in fact allows us to prove security in a stronger adversarial model where adversaries are allowed also to retrieve old states of parties or corrupt partial states such as group keys. For simplicity we do not consider such possible extensions of our model here.

1.4 Related Work

The basic idea of TreeKEM can be traced back to Logical Key Hierarchies (LKH) [24, 25, 7]. These were introduced as an efficient solution to multicast key distribution (MKD), where a trusted and central authority wants to encrypt messages to a dynamically changing group of receivers. Clearly, the main difference to continuous group key agreements is the the presence of a central authority that distributes the keys to users and may add and remove users. At the heart of TreeKEM is the realization that if one replaces symmetric key encryption with public key encryption in LKH, then any group member can perform the actions that the central authority does in MKD. But, as described above, this introduces the problem that some users now know the secret keys in parts of the tree they are not supposed to, which creates security problems. This is where the main novelties of TreeKEM and follow up work lies: in providing mechnanisms to achieve PCS and FS nonetheless.

LKH has been proven secure even against adaptive adversaries with a quasi-polynomial time bound [20]. Unfortunately, there are several important differences between LKH that do not allow us to simply rely on [20] to prove TTKEM or TreeKEM secure: 1) the proof of [20] is in the symmetric key setting, while we are using public key encryption; 2) the proof of [20] assumes a central authority and there is no concept of PCS or FS; 3) for efficiency reasons, TTKEM and TreeKEM use hierarchical key derivation, which the proof in [20] does not take into account (even though it had already been proposed optimized versions of LKH [7]) and it is a priori unclear how this affects the proof; 4) we are also interested in proving security in the random oracle model, which, as we show, gives tighter security bounds.

Since the appearance of the double ratchet algorithm [21], implemented in applications like Signal or Whatsapp, secure messaging has received a lot of attention, particularly in the two party case [4, 10, 16, 18, 22]. In the group setting, the main example of such a protocol is TreeKEM [6, 15], currently in development by the IETF MLS working group. Its predecesor was the ART protocol [8], whose proposal motivated the creation of the mentioned working group. A study of PCS in settings with multiple groups was done by Cremers *et al.*[9], and Weidner [19] explored a variant of TreeKEM allowing for less reliance on the server for correctness. All these protocols are related to broadcast encryption [11] and multicast encryption [20].

Alwen *et al.*[3] introduced a variant of TreeKEM, termed re-randomized TreeKEM (rTreeKEM), which significantly improves the FS guarantees that TreeKEM provides. Essentially, their construction can achieve FS for a party by just processing an update of another party, while (Tainted) TreeKEM requires a party to actively issue an update operation for that. Their work defines continuous group key agreement (CGKA) as an abstraction of the main problem TreeKEM aims to solve. We use their completeness notion for CGKA, but our security notion and bounds differ significantly. First, in order to prove security, [3] require the delivery server to be basically honest; the server can delay, but never send inconsistent messages to parties, while we allow the delivery server to be under adversarial control. Second, they prove security against selective adversaries with a polynomial security loss, and sketch a security proof against adaptive adversaries losing a quasi-polynomial factor (for TreeKEM in the standard model, for rTreeKEM in the ROM). We give a proof for adaptive security with only polynomial loss in the ROM and quasi-polynomial in the standard model for both TreeKEM and TTKEM.

It seems plausible that the proof techniques developed in this work can also be applied to the rTreeKEM construction or even a construction that combines the efficiency improvements of TTKEM with the improved FS of rTreeKEM.

2 Description of TTKEM

2.1 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* $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{keygen}(1^\lambda)$ *by users prior to joining a group. Public keys 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* $\mathsf{G} = (\mathsf{ID}_1, \dots, \mathsf{ID}_n)$*. Party* ID_1 *creates a new group with membership* G *by running:*

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

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* pending *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 the new pending state* 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 of (hopefully) making 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.

2.2 Overview

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. Definition 7). Note that paths in the tree now start at leaves and end at the root node. The nodes in the tree are associated with the following values: a seed $\varDelta$; a secret/public key pair derived deterministically as $(pk, sk) \leftarrow \mathsf{Gen}(\varDelta)$; a credential (leaf nodes only); and a tainted ID (all nodes except leaves and root). The root has no public/secret key pair associated with it, instead its seed is the current group key.

To achieve FS and PCS, and to manage group membership, it is necessary to constantly renew the secret keys used in the protocol. We will do this through the group operations **Update**, **Remove** and **Add**. 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. 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. As outlined in the introduction, users will also have a list of Initialization Keys (init keys) stored in some key-server, widely available and regularly updated, that will be used to add users to new groups.

Each group member 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. ID's that are part of the group); $\mathcal{T}$ denotes a binary tree as above, with each group member's their credential associated to a leaf node; and $\mathcal{H}$ denotes the hash of the group transcript so far, to ensure consistency.

Each user also has a, typically empty, *pending state* $\gamma'(\mathsf{ID})$ which stores the changes done by the last Add, Remove or Update operation issued by them 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, as in TreeKEM, we would ideally like the following predicate to hold:

A user knows the secret key associated to a node v if and only if v 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 is *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. This can be empty, i.e. the node is untainted, or corresponds to a single party's ID, that who last generated this node's secret but is not supposed to know it. The tainter ID of a node is determined by the following simple rules:

– After ID initialises, all internal nodes not on ID's path become tainted by ID.
– If ID updates, the nodes on ID's path become untainted.
– If ID updates, all refreshed nodes *not* on ID's path become tainted by ID.

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 follow the specification of the MLS draft [15], where more details can be found. Essentially we consider a hash function H, fix two tags x_1 and x_2 and consider the two hash functions H_1, H_2 with $H_i(\cdot) = H(\cdot, x_i)$. Together with a Gen function that outputs a secret-public key pair, we derive the keys for the nodes asfollows:

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

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. For the proof in the standard model we only require H_i to be pseudorandom functions, with Δ_i the key and x_i the input.

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 partition all the nodes to be refreshed into paths and use a different seed for each path. For this we need a unique path cover, as users processing the update will need to know which nodes secrets depend on which. A concrete example is given in section A.2, but any unambiguous partition suffices. The only condition required is that the updating of paths is done in a particular common order that allows for encryptions to to-be-refreshed nodes to be done under the respective updated public key (one cannot hope for PCS otherwise).

Let us stress that a party processing an update 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.

2.3 TTKEM Dynamics

Whenever a user ID_i wants to perform a group operation, she will generate the appropriate Initialize, Update, Add or Remove message, store the updated state resulting from processing such message in γ', and send the

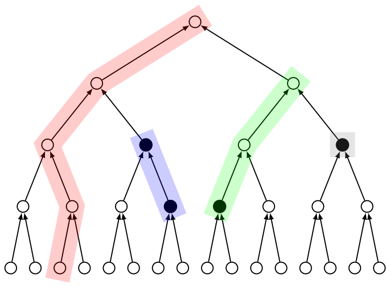


Fig. 2: Schematic diagram showing the path partition for an update done by Charlie (3rd leaf node), where the black nodes are tainted by him. A user would need to update the grey node before the green path and the blue path before the red (Charlie's) path.

appropriate information to the delivery server, which will then respond with a confirm or reject, prompting ID_i to move to state γ' (i.e. set $\gamma \leftarrow \gamma'$) or to delete γ' respectively. If the (honest) delivery server confirms an operation it will also deliver it to all the group members, who will process it and update their states accordingly. Messages should contain the identity of the sender, the type of operation, encryptions of the new seeds, any new public keys, and a hash of the transcript to far, to ensure consistency. The detailed content of these messages, together with pseudo-code for the distinct operations is presented in A.3.

Initialize. To create a new group with parties $\mathcal{M} = \{\mathsf{ID}_1, \ldots, \mathsf{ID}_n\}$, a user ID_1 generates a new tree $\mathcal{T}$, where the leaves have associated the init keys corresponding to the group members. The group creator then samples new key pairs for all the other nodes in $\mathcal{T}$ (optimizing with hierarchical derivation) and crafts welcome messages for each party. These welcome messages should include an encryption of the seed that allows the computation of the keys of the appropriate path, together with $\mathcal{M}$ and the public part of $\mathcal{T}$.

Add. To add a new member ID_j to the group, ID_i identifies a free spot for them, hashes her secret key together with some freshly sampled randomness to obtain a seed Δ[12], and derives seeds for the nodes along the path to the root. She then encrypts the new seeds to all the nodes in the co-path (one ciphertext per node suffices given the hierarchical derivation) and sends them over together with the identity ID_j of the added party. ID_i will also craft a welcome message for the added party containing an encryption of the appropriate seed, $\mathcal{M}$, $\mathcal{H}$ and the public part of $\mathcal{T}$.

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

Remove. To remove a user ID_j, user ID_i performs an Update as if it was ID_j, refreshing all the nodes in ID_j's path to the root, as well as all her tainted nodes (which will become tainted by ID_i after the removal). Note that a user cannot remove itself. Instead, we imagine a user that wants to leave the group could request for someone to remove her and delete her state.

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, by updating the list of participants if necessary, overwriting any keys, and updating the tainted ID's. If it is a confirm or a reject, i.e. it was an operation issued by himself, it updates the current state γ to be the pending state γ' and deletes the pending state in the prior case; or simply deletes the pending state in the latter.

[12] This way the new keys will be secure against an adversary that does not have either knowledge of ID_i's secret key or control/knowledge of the randomness used.

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. However, as mentioned before, tainting seems to be a more natural approach: it maintains the desired tree structure, and its bookkeeping method gives us a more complete intuition of the security of the tree. It also corresponds to a more flexible framework: since blanking simply forbids parties to know secrets outside of their path, it 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 in the set of affected users: a blank node degrades the efficiency of *any user* whose copath contains the blank. Conversely, a tainted node affects only *one* user; that 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 refreshes it (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 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.

To test this, we ran simulations comparing the number of ciphertexts (cost) users need to compute on average as a consequence of performing Updates, Adds and Removes. Ideally, we would like to sample a sequence of group operations, execute them in both protocols and compare the total cost. However, this seems infeasible: in TreeKEM operations are collected into Commits, whereas in TTKEM these are applied one by one, separately. Hence, we compared TTKEM (referred to as *tainted* in the graphs) against two different simplified versions of TreeKEM, between which real TreeKEM lies efficiency-wise. The first version (*TKEM*), more efficient than actual TreeKEM, ignores Commits and just executes operations one by one, without the Update that would follow every Commit. The second version (*TKEM_commit*), less efficient than the real TreeKEM, enforces that every operation is committed separately, essentially performing an extra Update operation after every Add or Remove.

We simulated groups of sizes between 2^3 and 2^{15} members. Trees of size 2^i were initialized with 2^{i-1} members and sequences of $10*2^i$ Update/Remove/Add operations were sampled according to a $8:1:1$ ratio. One would expect for many more Updates than Add/Removes to take place; but also, the more common updates are, the closer that efficiency is going to be to that of naïve TreeKEM for both TreeKEM and TTKEM. Thus, this seems a reasonable ratio that also highlights the differences between the protocols - it is also the ratio used by R. Barnes in the simulation of TreeKEM with blanking posted in the IETF MLS mailing list[13]. We test two different scenarios. In the first one we limit the ability of adding and removing parties to a small subset of users, the administrators. In the second, we make no assumption on who performs Adds and Removes and sample the authors of these uniformly at random.

To simulate the administrator setting (figures 3, 4 and 5), we set a small (1 per group in groups of size less tan 128 and 1 per every 64 users in bigger groups) random set of users to be administrators. Adds and Removes are then performed by one of those administrators sampled uniformly at random. The removed users, as well as the authors of the updates were also sampled uniformly at random. Figures 3 and 4 illustrate that TTKEM allows for an interesting trade-off, where non-administrators enjoy more efficient Updates at the expense of potentially more work for administrators. This would be favourable in settings

[13] [MLS] Cost of the partial-tree approach. Richard Barnes {rlb@ipv.sx} 01 October 2018 `https://mailarchive.ietf.org/arch/msg/mls/hhl0q-0gnGUJS1djdmH1JBMq0SY/`

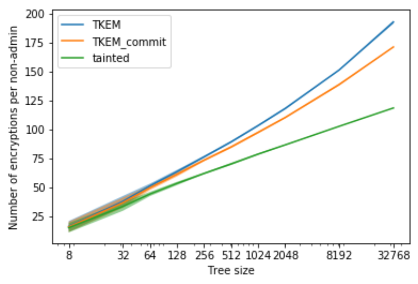


Fig. 3: Cost for non-administrators

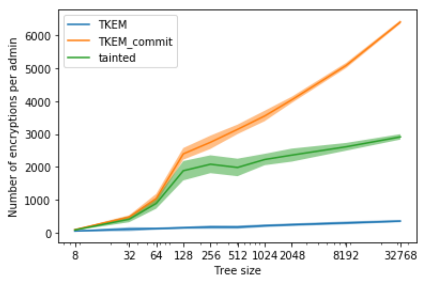


Fig. 4: Cost for administrators

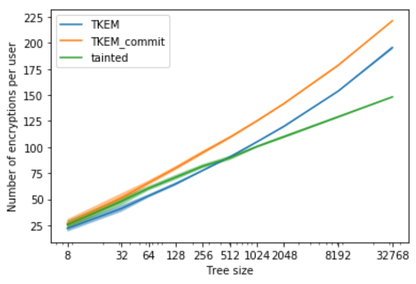


Fig. 5: Average cost per user

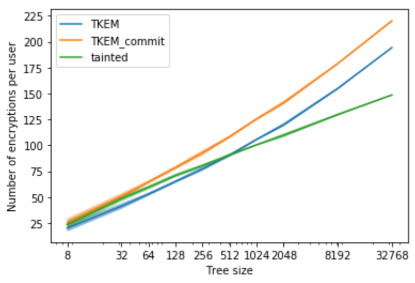


Fig. 6: Updaters follow uniform dist.

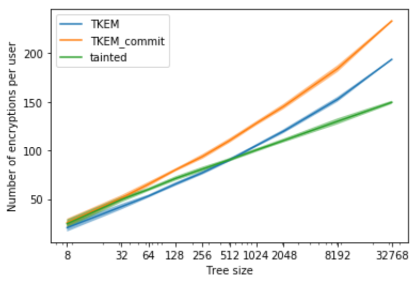


Fig. 7: Updaters follow Zipf dist.

where administrators have more bandwidth or computational power. When considering the average cost incurred by group member, admins or non-admins (figure 5), all three protocol perform similarly for smaller groups, with TTKEM behaving better asymptotically.

In the second scenario (figures 6 and 7), where Adds and Removes are performed by users sampled uniformly, the results are similar: all protocols perform comparably on smaller groups, with TTKEM behaving more efficiently on larger groups. Here, we distinguish two further situations depending on the distribution on Update authors (or *updaters* for short). Figure 6 shows the results of sampling updaters uniformly at random. This would reflect scenarios where Updates are executed periodically, as in e.g. devices that are always online and where a higher level policy stipulates to update daily. In contrast, figure 7 shows the results of sampling updaters following a Zipf distribution. The Zipf distribution is used widely to model human activity in interactive settings. Recently, a study on messages sent on internet communities shows that the growth of messages sent per individual over time follows Gibrat's law [23]. This in turn implies that the distribution of the number of messages sent per individual at a point in time converges asymptotically to a Zipf distribution [14]. Thus, the latter scenario models a setting where Updates are correlated with the level of activity of the users, e.g. when the devices used are not always online.

Overall, while we cannot say TTKEM will be more efficient than TreeKEM in every setting, it is clear that it constitutes a promising CGKA candidate, which can bring efficiency improvements over TreeKEM in different realistic scenarios. Moreover, we would also like to point out that to improve the efficiency of these protocols, different policies can be implemented, such as strategically placing users on the tree, e.g. distributing administrators or frequent updaters closer to the right side of the tree, where more new users will be added.

Finally, since our simulations above are initialized with a fully healed tree (no blanks or taints), we discuss the different costs of the initialization phase, i.e. the process of creating a group and clearing all blanks or taints, in both protocols. We find that TTKEM can achieve such a healed tree considerably faster than TreeKEM.

Efficiency of Initialisation. For many group chat sessions, the initialisation phase will be the most inefficient phase of the session's life-cycle, as 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) of transitioning to a fully healed ratchet tree in the least number of group operations (without further Adds, Removes or redundant Updates).

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.[14] 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, meaning we can achieve a fully untainted tree with a total communication cost of $(\lfloor n/2 \rfloor - 1) \log n$. This is not the case with blanking, where the order matters. The following lemma shows that the cost for blanking is at least about twice as much as with tainting. We defer its proof to B.

Lemma 1. *In TreeKEM, the cost of transitioning from a completely blank (except for the creator's path) tree to one without blanks is at least* $n (\log n - 2) + 2$*.*

3 Security

We will prove security for TTKEM against fully adaptive, partially active adversaries, even when group members are in inconsistent states. In section 3.1 we present the security game we consider and in section 3.2 we present a simple predicate which allows to determine for which group keys we can guarantee security. The latter predicate incorporates the intuition that Updates allow a party to heal her state. It should be noted that we consider initialization keys as representing identities, as otherwise we would neglect some other cases which we would intuitively also consider secure, such as removing a corrupted party and adding them again once uncorrupted (this is secure per our predicate as they would be treated as a new identity, generated at the time the init key was).

Throughout our proofs, we only consider a single challenge per game for simplicity; a standard hybrid argument allows us to extend security to multiple challenges, with a loss linear in the number of challenges. In order to simulate extra challenges, an extra oracle that reveals group keys would be needed, but this would have no effect on the security proof - in particular GSD-like proofs already allow for the corruption of individual keys.

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**(G) *and the challenger 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** *allow 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** *enable the adversary to corrupt the users for a time period. The entire state (old and pending) and random coins of a corrupted user are leaked to the adversary during this period.*

[14] In particular, the less bandwidth used per Update the more Updates can be performed and so the stronger the expected security properties 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 current local state* γ*.*
4. **confirm**(q, β)*: the q-th query in the game, which must be an action* $\mathsf{a} \in \{$**add-user**, **remove-user**, **update**$\}$ *by some user* ID*, is either confirmed (if* $\beta = 1$*) or rejected (if* $\beta = 0$*). In case the action is confirmed,* C *updates* ID*'s state and deletes the previous state; otherwise* ID *keeps its previous 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 internal state and randomness of* ID *is leaked to the adversary.*
7. **end-corrupt**(ID)*: ends the leakage of user* ID*'s internal state and randomness to the adversary.*
8. **challenge**(q^*)*:* A *picks a query* q^* *corresponding to an action* $\mathsf{a}^* \in \{$**add-user**, **remove-user**, **update**$\}$ *or the initialization (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**$(\cdot,\cdot)$, **remove-user**$(\cdot,\cdot)$*, or* **update**$(\cdot)$ *and running in time* t *it holds*

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

3.2 The Safe Predicate

We define the *safe predicate* to rule out trivial winning strategies and at the same time restrict 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[15] 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 is very subtle.

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 PCS, any corruption of ID^* *before* q^* should not harm security. Similarly, any corruption of ID^* *after* a further processed update operation for ID^* should not help the adversary either (compare FS). Finally, also in the case where the update generated at time q^* is rejected to ID^* and ID^* processes this message of the form **confirm**$(q^*, 0)$ 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.

Additionally, we 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. As mentioned above, 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: 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^*; 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 careful here, since in a similar scenario where ID does not process the reject for its own update, the challenge group key would clearly not be safe anymore.

[15] 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.

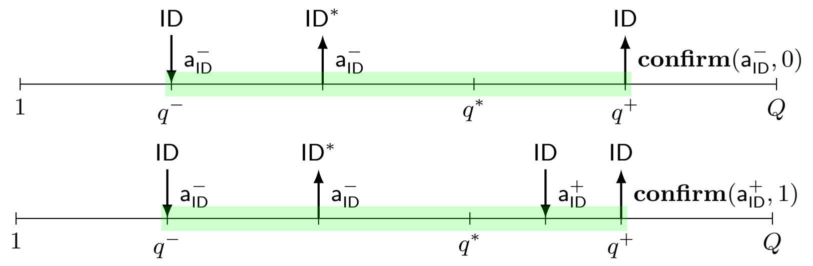
For the following definitions we consider discrete time steps measured in terms of the number of queries that have been issued by the adversary so far.

We first identify for each user a critical window in the view of a specific user ID^*. The idea is to define exactly the time frame in which a user may leak a group key if ID^* generates it at a specific point in time and distributes it to the group. Clearly, the users may not be corrupted in this time frame if this happens to be the challenge group key.

Definition 3 (Critical window, safe user). *Let ID and ID^* be two (not necessarily different) users and $q^* \in [Q]$ be some query. Let $q^- \leq q^*$ be the query that set ID's current key* in the view of ID^* at time q^*, *i.e. the query $q^- \leq q^*$ that corresponds to the last update message $\mathsf{a}_{\mathsf{ID}}^- := \mathbf{update}(\mathsf{ID})$ processed by ID^* at some point $[q^-, q^*]$ (see Figure 8). If ID^* does not process such a query then we set $q^- = 1$, the first query. Analogously, let $q^+ \geq q^-$ be the first query that invalidates ID's current key, i.e. ID processes one of the following two confirmations:*

1. $\mathbf{confirm}(\mathsf{a}_{\mathsf{ID}}^-, 0)$*, the rejection of action $\mathsf{a}_{\mathsf{ID}}^-$; or*
2. $\mathbf{confirm}(\mathsf{a}_{\mathsf{ID}}^+, 1)$*, the confirmation an update* $\mathsf{a}_{\mathsf{ID}}^+ := \mathbf{update}(\mathsf{ID}) \neq \mathsf{a}_{\mathsf{ID}}^-$.

If ID does not process any such query then we set $q^+ = Q$, the last query. We say that the window $[q^-, q^+]$ is critical *for ID at time q^* in the view of ID^*. Moreover, if the user ID is* not corrupted *at any time point in the critical window, we say that ID is* safe *at time q^* in the view of ID^*.*



ID
ID*
ID
$\mathsf{a}_{\mathsf{ID}}^-$
$\mathsf{a}_{\mathsf{ID}}^-$
$\mathbf{confirm}(\mathsf{a}_{\mathsf{ID}}^-, 0)$
1
q^-
q^*
q^+
Q
ID
ID*
ID
ID
$\mathsf{a}_{\mathsf{ID}}^-$
$\mathsf{a}_{\mathsf{ID}}^-$
$\mathsf{a}_{\mathsf{ID}}^+$
$\mathbf{confirm}(\mathsf{a}_{\mathsf{ID}}^+, 1)$
1
q^-
q^*
q^+
Q

Fig. 8: A schematic diagram showing the critical window for a user ID in the view of another user ID^* with respect to query q^*. An arrow from a user to the timeline is interpreted as a request by the user, whereas an arrow in the opposite direction is interpreted as the user processing the message. The figure at top (resp., bottom) corresponds to the first (resp., second) case in Definition 3.

We are now ready to define when a *group key* should be considered *safe*. The group key is considered to be safe if all the users that ID^* considers to be in the group are individually safe, i.e., not corrupted in its critical window, in the view of ID^*. We point out that there is a exception when the action that generated the group key sk^* is a self-update by ID^* where, to *allow healing*, instead of the normal critical window we use the window $[q^*, q^+]$ as critical.

Definition 4 (Safe predicate). *Let sk^* be a group key generated in an action*

$$\mathsf{a}^* \in \{\mathbf{add\text{-}user}(\mathsf{ID}^*, \cdot), \mathbf{remove\text{-}user}(\mathsf{ID}^*, \cdot), \mathbf{update}(\mathsf{ID}^*), \mathbf{create\text{-}group}(\mathsf{ID}^*, \cdot)\}$$

at time point $q^* \in [Q]$ 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 ID^*. Then the key sk^* is considered safe *if for all users $\mathsf{ID} \in G^*$ (including ID^*) we have that ID is safe at time q^* in the view of ID^* (as per Definition 3) with the following exceptional case: if $\mathsf{ID} = \mathsf{ID}^*$ and $\mathsf{a}^* = \mathbf{update}(\mathsf{ID}^*)$ then we require ID^* to be safe with respect to the window $[q^*, q^+]$.*

3.3 The Challenge Graph

In the last section, we defined what it means for a group key to be safe via a *safe predicate*. In this section, we try to interpret the safe predicate for the TTKEM protocol. That is, our goal is to show that if the safe predicate is satisfied for a group key Δ^* generated while playing the CGKA game on TTKEM, then none of the seeds or secret keys *used to derive* this group key are leaked to the adversary (Lemma 2) — this fact will be crucial in the next section (§3.5) where we argue the security of TTKEM using the framework of Jafargholi et al. [17]. To this end, we view the CGKA game for TTKEM as a game on a graph and then define the *challenge graph* for challenge group key Δ^* as a sub-graph of the whole CGKA graph.

The CGKA graph. A node i in the CGKA graph for TTKEM is associated with seeds Δ_i and $s_i := H_2(\Delta_i)$ and a key-pair $(pk_i, sk_i) := \mathsf{Gen}(s_i)$ (as defined in §2). The edges of the graph, on the other hand, are induced by dependencies via the hash function H_1 or (public-key) encryptions. To be more precise, an edge (i, j) might correspond to either:

1. a ciphertext of the form $\mathsf{Enc}_{pk_i}(\Delta_j)$; or
2. an application of H_1 of the form $\Delta_j = H_1(\Delta_i)$ used in hierarchical derivation.

Naturally, the structure of the CGKA graph depends on the **update**, **add-user** or **remove-user** queries made by the adversary, and is therefore generated adaptively.

The challenge graph. The challenge graph for Δ^*, intuitively, is the sub-graph of the CGKA graph induced on the nodes from which Δ^* is trivially derivable. Therefore, according to the definition of the CGKA graph, this consists of nodes from which Δ^* is reachable and the corresponding edges (used to reach Δ^*). For instance, in the case where the adversary maintains all users in a consistent state and there are no tainted nodes, the challenge graph would simply be the binary tree rooted at Δ^* with leaves corresponding to init keys of users in the group at that point. When the group view is inconsistent among the users these leaves would correspond to the init keys of users in the view of ID^*. Moreover, if there are tainted nodes, the tree could also have (non-init key) leaves corresponding to these tainted nodes. Below we state and prove the key lemma which connects the safe predicate to the challenge graph of TTKEM.

Lemma 2. *For any* safe *challenge group key in TTKEM it holds that none of the seeds and secret keys in the challenge graph is leaked to the adversary via corruption.*

Proof. We first observe that in the CGKA game, when a user is corrupted all secret keys and seeds in its memory are also corrupt and therefore leaked to the adversary. This could consist of the secret keys and seeds on the *current* path from the user's leaf to the root (in the tree as currently viewed by the user itself), secret keys on a *pending* path (if one exists) and, by definition, all tainted nodes (as viewed by any user). Let's consider the challenge graph of any group key Δ^* generated by a user ID^* during an action

$$\mathsf{a}^* \in \{\textbf{add-user}, \textbf{remove-user}, \textbf{update}, \textbf{create-group}\}$$

at a time point q^*. Intuitively, the safe predicate for TTKEM ensures that *none* of the keys in this graph is leaked to the adversary by requiring that every node has been refreshed (via an update) before the generation of Δ^*. Therefore, information on Δ^* (via ciphertexts or hash dependencies) are sent *only* to refreshed and therefore uncorrupt nodes. Since the rest of the simulation in the CGKA game is independent of Δ^*, an adversary learns nothing about it. We formalise this intuition below via a proof by contradiction: assuming Δ^* is leaked to the adversary, we show that the safe predicate for Δ^* is somehow violated in the view of ID^* at q^*.

Let's first consider a restricted adversary that does not issue reject queries (i.e, $\textbf{confirm}(\cdot, 0)$). The fact that Δ^* is leaked to the adversary means that some secret key denoted sk in its challenge graph was leaked during the CGKA game (the argument when a seed s or Δ is leaked is similar). This in turn means that some user was corrupted while having sk in its memory. There are two possibilities: either sk corresponds to a leaf node and is a user key for a user ID in the group (possibly same as ID^*) (Case A) or sk corresponds to an internal node (Case B). Let's argue these cases separately.

Case A. Let's suppose first that $\mathsf{ID} \neq \mathsf{ID}^*$. As sk is leaked to the adversary it follows that ID was corrupted *after* ID generated the update ($\mathsf{a}^-_{\mathsf{ID}}$) and *before* ID processed the update $\mathsf{a}^+_{\mathsf{ID}} \neq \mathsf{a}^-_{\mathsf{ID}}$ (see Figure 9, top). The reason being that this is the *only* window in which this key is present in the memory of ID (and this key is *not* present in the memory of any other user). However, this violates the safe predicate (restricted to the case where there are no rejects) as ID is not a safe user. In the complementary case where $\mathsf{ID} = \mathsf{ID}^*$, the argument is similar except that the window in which sk is present in the memory is different (as in the exceptional case in Definition 4).

Case B. When sk is part of an internal node, we apply the argument in Case A to every user that is an ancestor of (the node) sk in the challenge graph: it would then follow that at least one of these users violates the safe predicate. However, if not every leaf in the set of ancestors of (the node) sk in the challenge graph correspond to a particular user, which can *only* happen when it corresponds to a tainted node, we apply this argument to the owner of the tainted node. Note that, by construction, the owner must be a part of the group as viewed by ID^* (even if the states are inconsistent) as all tainted nodes corresponding to a removed user are also removed from the state.

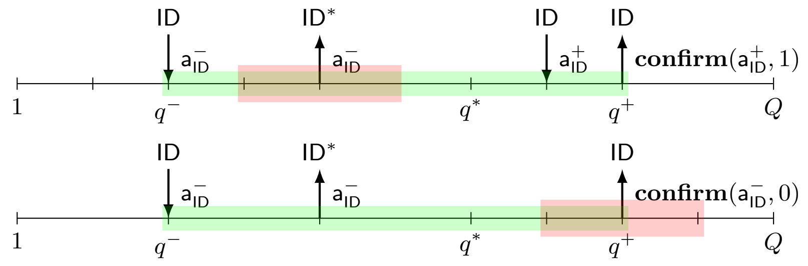

ID
ID*
ID
ID
$\mathsf{a}^-_{\mathsf{ID}}$
$\mathsf{a}^-_{\mathsf{ID}}$
$\mathsf{a}^+_{\mathsf{ID}}$
confirm$(\mathsf{a}^+_{\mathsf{ID}}, 1)$
1
q^-
q^*
q^+
Q
ID
ID*
ID
$\mathsf{a}^-_{\mathsf{ID}}$
$\mathsf{a}^-_{\mathsf{ID}}$
confirm$(\mathsf{a}^-_{\mathsf{ID}}, 0)$
1
q^-
q^*
q^+
Q

Fig. 9: Violation of the safe predicate. The critical window for ID are shaded in green, and the windows where ID was corrupt is shaded in red. In all the cases, there is an intersection between the critical and corrupt window and therefore the safe predicate is violated.

Now, in the case where the adversary does reject queries, the argument is similar except that we have to take into account the case where a user key sk is invalidated by such rejects. To be precise, in the case where $\mathsf{a}^-_{\mathsf{ID}}$ is rejected to ID it could be the case that the safe predicate is violated when a corruption occurs before $\mathsf{a}^-_{\mathsf{ID}}$ is rejected to ID (see Figure 9, bottom). Finally, note that in the above argument it was implicitly assumed that all users corresponding to the leaf nodes of Δ^* were part of the group G^* viewed by ID^* at time q^* if the query q^* was processed. This is crucial since the safe predicate is applied *only* to the users in G^*. ☐

3.4 Security Proof for TTKEM in the Standard Model

To prove security of TTKEM in the standard model, we will use the framework of Jafargholi et al. [17]. Recall that in the CGKA security game, the aim of the adversary is to distinguish a safe challenge group key Δ^* from a uniformly random and independent seed. We first consider the *selective* CGKA game, where the adversary has to do all its queries at once. We call the two possible executions of the game the *real* and *random* CGKA game and aim to proof indistiguishability of these two games via a sequence of indistinguishable hybrid games. Similar to several other applications of the framework [17], we will define these hybrid games via the so-called *reversible black pebbling* game, introduced by Bennett [5], where, given a directed acyclic graph with unique sink (here, the challenge graph), in each step one can put or remove one pebble on a node following certain rules, and the goal is to reach the pebbling configuration where there is only one pebble on the sink of the graph. Each *pebbling configuration* $\mathcal{P}_\ell$ then uniquely defines a *hybrid game* H_ℓ: a node v in the tree being pebbled means that in this hybrid game whenever Δ_v would be used to answer a query, a freshly chosen random seed (independent of Δ_v) is used instead in the simulation. This applies to all cases where Δ_v would be used as input for H_1 or H_2, or as the challenge output (if i is the challenge node). All remaining nodes and edges are simulated as in the real CGKA game. Thus, the real game $\mathsf{H}_{\mathsf{real}}$ is represented as the empty pebbling configuration $\mathcal{P}_0$ where there is no pebble at all, while the random game $\mathsf{H}_{\mathsf{random}}$ corresponds to the final configuration $\mathcal{P}_L$ where only the sink node is pebbled (where L denotes the length of the pebbling sequence).

Definition 5 (Reversible black pebbling). *A reversible pebbling of a directed acyclic graph $G = (V, E)$ with unique sink sink is a sequence $\mathfrak{P} = (\mathcal{P}_0, \ldots, \mathcal{P}_L)$ with $\mathcal{P}_\ell \subset V$ ($\ell \in [0, L]$), such that $\mathcal{P}_0 = \emptyset$ and $\mathcal{P}_L = \{\mathsf{sink}\}$, and for all $\ell \in [L]$ there is a unique $v \in V$ such that: 1) $\mathcal{P}_\ell = \mathcal{P}_{\ell-1} \cup \{v\}$ or $\mathcal{P}_\ell = \mathcal{P}_{\ell-1} \setminus \{v\}$, 2) for all $u \in \mathrm{parents}(v)$: $u \in \mathcal{P}_{\ell-1}$.*

By Lemma 2, we know that none of the seeds or secret keys in the challenge graph is leaked to the adversary throughout the entire game. This will allow us to prove indistinguishability of subsequent hybrid games from IND-CPA security of the underlying encryption scheme and pseudorandomness of the hash functions H_1, H_2. Recall, the functions H_1, H_2 were defined by a hash function H which takes some Δ_i as secret key and publicly known fixed strings x_1, x_2 as inputs. To guarantee security, H is assumed to be a pseudorandom function, where we will use the following non-standard but equivalent (to the standard) definition of pseudorandomness:

Definition 6 (Pseudorandom function, alternative definition). *Let $H : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be a keyed function. We define the following game $\mathsf{PRF}(n)$: First, a key $k \leftarrow \{0,1\}^n$ is chosen uniformly at random and the adversary is given access to an oracle $H(k, \cdot)$. When the adversary outputs a string $x \leftarrow \{0,1\}^n$, a uniformly random bit $b \leftarrow \{0,1\}$ is chosen and the adversary receives either $H(k, x)$ in the case $b = 0$, or $y \in \{0,1\}^n$ uniformly at random if $b = 1$. Finally, the adversary outputs a bit b'. If x was never queried to the oracle $H(k, \cdot)$ and $b' = b$, then the output of the game is 1, otherwise 0. We call H* (ϵ, t)-pseudorandom *if for all adversaries A running in time t we have*

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

It is an easy exercise to prove that the above definition is equivalent to the standard textbook definition of pseudorandom functions (i.e., only a polynomial loss in security is involved by the respective reductions).

Lemma 3. *Let $\mathfrak{P} = (\mathcal{P}_0, \ldots, \mathcal{P}_L)$ be a valid pebbling sequence on the challenge graph. If H is an (ϵ, t)-secure pseudorandom function and $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is an (ϵ, t)-IND-CPA secure encryption scheme, then any two subsequent hybrid games $\mathsf{H}_\ell, \mathsf{H}_{\ell+1}$ are $(5 \cdot \epsilon, t)$-indistinguishable*[16].

Proof. Let $\mathsf{H}_\ell, \mathsf{H}_{\ell+1}$ be two subsequent hybrid games. We assume that $\mathcal{P}_{\ell+1}$ differs from $\mathcal{P}_\ell$ by one additional pebble on a node v^* with ingoing encryption edge (u, v^*) and ingoing H_1 edge (u', v^*). The case where $\mathcal{P}_{\ell+1}$ is obtained from $\mathcal{P}_\ell$ by removing one pebble can be proven in a similar way. We proof indistinguishability by a sequence of hybrids $\mathsf{H}_\ell := \mathsf{H}_{\ell,0}, \mathsf{H}_{\ell,1}, \ldots, \mathsf{H}_{\ell,5} := \mathsf{H}_{\ell+1}$, where the intermediate hybrids are defined as follows:

– $\mathsf{H}_{\ell,1}$ is defined similarly to $\mathsf{H}_{\ell,0}$ except that the key pair $(\mathsf{pk}_u, \mathsf{sk}_u)$ is generated from a uniformly random seed instead of the output s_u of H_2.
– $\mathsf{H}_{\ell,2}$ is defined similarly to $\mathsf{H}_{\ell,1}$ except that the encryption $\mathsf{Enc}_{\mathsf{pk}_u}(\Delta_{v^*})$ is replaced by an encryption of a uniformly random seed.
– $\mathsf{H}_{\ell,3}$ is defined similarly to $\mathsf{H}_{\ell,2}$ except that instead of Δ_{v^*} a uniformly random seed is used as input to H_1, or H_2, or the challenge output (if v^* is the challenge node) whenever needed to answer any queries of the adversary.

[16] Technically, the t in Lemma 3 changes slightly due to the reduction and thus should not actually be the same t. For simplicity, in all our security reductions we will ignore such miniscule running time overheads incurred by simulating challengers of the security games or sampling (a small number of) random bits.

– $\mathsf{H}_{\ell,4}$ is defined similarly to $\mathsf{H}_{\ell,3}$ except that the encryption of a uniformly random seed is replaced by $\mathsf{Enc}_{\mathsf{pk}_u}(\Delta_{v^*})$.
– $\mathsf{H}_{\ell,5}$ is defined similarly to $\mathsf{H}_{\ell,3}$ except that the key pair $(\mathsf{pk}_u, \mathsf{sk}_u)$ is generated from $s_u = H_2(\Delta_u)$ again, instead of using a uniformly random seed. Note that indeed $\mathsf{H}_{\ell,5} = \mathsf{H}_{\ell+1}$ holds.

Indistinguishability of $\mathsf{H}_{\ell,0}$ and $\mathsf{H}_{\ell,1}$ follows from the pseudorandomness of the hash function H: Since by the pebbling rules it must hold that node u is pebbled, in game $\mathsf{H}_{\ell,0}$ the seed s_u for key generation is computed by applying H_2 to a uniformly random seed instead of Δ_u. Thus, by pseudorandomness of H the seed s_u is indistinguishable from a uniformly random seed, as used in $\mathsf{H}_{\ell,1}$. Furthermore, all the remaining seeds and edges needed during simulation of the hybrid games can be perfectly simulated, which implies that any advantage ϵ of an adversary in distinguishing $\mathsf{H}_{\ell,0}$ from $\mathsf{H}_{\ell,1}$ leads to the same advantage ϵ in distinguishing H from a random function.

Indistinguishability of $\mathsf{H}_{\ell,1}$ and $\mathsf{H}_{\ell,2}$ follows from the IND-CPA security of the encryption scheme: Since in game $\mathsf{H}_{\ell,1}$ the key pair $(\mathsf{pk}_u, \mathsf{sk}_u)$ is generated just as in Gen, a reduction can embed an IND-CPA challenge for messages Δ_{v^*} and a uniformly random seed at the encryption edge (u, v^*) and otherwise perfectly simulate all answers to queries of the adversary. Thus, any adversary distinguishing $\mathsf{H}_{\ell,1}$ from $\mathsf{H}_{\ell,2}$ with advantage ϵ can be used to break IND-CPA security of the encryption scheme with the same advantage.

Indistinguishability of $\mathsf{H}_{\ell,2}$ and $\mathsf{H}_{\ell,3}$ follows from the pseudorandomness of the hash function H: This is similar to the case of $\mathsf{H}_{\ell,0}$ and $\mathsf{H}_{\ell,1}$, and the indistinguishability of Δ_{v^*} again cruicially relies on the fact that node u' is pebbled. However, the alternative definition of pseudorandomness must be used for the reduction, since the seed $s_{u'}$ used to generate the key pair $(\mathsf{pk}_{u'}, \mathsf{sk}_{u'})$ is the output of H with the same random seed as used to compute Δ_{v^*}. Again, the remaining simulation can be done perfectly and the reduction preserves the advantage.

Similar to the case of $\mathsf{H}_{\ell,1}$ and $\mathsf{H}_{\ell,2}$, indistinguishability of $\mathsf{H}_{\ell,3}$ and $\mathsf{H}_{\ell,4}$ again follows from the IND-CPA security of the encryption scheme.

Finally, indistinguishability of $\mathsf{H}_{\ell,4}$ and $\mathsf{H}_{\ell,5}$ again follows from pseudorandomness of the hash function H, similar to the case of $\mathsf{H}_{\ell,0}$ and $\mathsf{H}_{\ell,1}$.

Note that in all the sketched reductions, the running time remains essentially the same. Thus, the claim follows. □

Choosing a trivial pebbling sequence of the challenge graph, this already implies *selective* CGKA security of TTKEM. Unfortunately, in the adaptive setting, the challenge graph is not known to the reduction until the adversary does its challenge query, but by this time it will be too late for the reduction to embed a challenge, since seeds and public keys in the challenge graph might have been used already before when answering previous queries by the adversary. Thus, to simulate a hybrid game H_ℓ, the reduction needs to *guess* (some of) the adaptive choices the adversary will do. Naïvely, this would result in an exponential loss in security. However, the framework of Jafargholi et al. [17] allows us to do significantly better:

Theorem 1 (Framework for proving adaptive security, informal [17]). *Let $\mathsf{G}_{\mathsf{real}}$, $\mathsf{G}_{\mathsf{random}}$ be two adaptive games, and $\mathsf{H}_{\mathsf{real}}$, $\mathsf{H}_{\mathsf{random}}$ be their respective selective versions, where the adversary has to do all its choices right in the beginning of the game. Furthermore, let $\mathsf{H}_{\mathsf{real}} := \mathsf{H}_0, \mathsf{H}_1, \ldots, \mathsf{H}_L := \mathsf{H}_{\mathsf{random}}$ be a sequence of hybrid games such that each pair of subsequent games can be simulated and proven (ϵ, t)-indistinguishable by* only guessing M bits of information on the adversary's choices. *Then $\mathsf{G}_{\mathsf{real}}$ and $\mathsf{G}_{\mathsf{random}}$ are $(\epsilon \cdot L \cdot 2^M, t)$-indistinguishable.*

The problem of proving CGKA security of TTKEM now reduces to finding a sequence of indistinguishable hybrids such that each hybrid can be simulated by only a small amount of random guessing. Defining hybrid games via pebbling configurations as above and using the space-optimal pebbling sequence for binary trees, described e.g. in [13, Algorithm 1], which uses $L = n^2$ steps and only $2\log(n)+1$ pebbles[17], implies a security reduction for TTKEM with only a quasipolynomial loss in security.

[17] Although the original Lemma 3 in [13] states that $3\log(n)$ pebbles are required to pebble a binary tree, the bound is loose since it is derived from Lemma 2. It is not difficult to see that a tighter analysis of Algorithm 1 for the case of binary tree leads to a bound of $\log 2 + 1$.

Theorem 2. *If H is an (ϵ, t)-pseudorandom function and $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is an (ϵ, t)-IND-CPA secure encryption scheme, then TTKEM is $(5 \cdot n^2 \cdot Q^{\log(n)+2} \cdot \epsilon, t)$-GCKA secure.*

Proof. Note that the challenge graph is a complete binary tree of depth $\log(n)$ in the worst case and let $\mathfrak{P} = (\mathcal{P}_0, \ldots, \mathcal{P}_L)$ be the recursive pebbling strategy for binary trees from [13], which uses $L = n^2$ steps and at most $2\log(n) + 1$ pebbles. We will prove that each pebbling configuration $\mathcal{P}_\ell$ can be represented using $M = (\log(n) + 1) \cdot \log Q$ bits. The claim then follows by Lemma 3 and Theorem 1.

We need the following property of the strategy $\mathfrak{P}$: For all $\ell \in [0, L]$, there exists a leaf in the tree such that all pebbled nodes lie either on the path from that leaf to the sink or on the copath. Furthermore, the subgraph on this set of potentially pebbled nodes contains $2\log(n) + 1$ nodes which are connected by at most $\log(n) + 1$ encryption and H_1 edges, respectively. Throughout the game, the reduction always knows in which position in the binary tree a node ends up, but it does not know which of the up to Q versions of the node will end up in the challenge tree. However, nodes connected by an H_1 edge are generated at the same time, so the reduction only needs to guess for at most $\log(n) + 2$ nodes which of the up to Q versions of that node will be in the challenge graph. This proves the claim. $\square$

Since the above proof mainly relies on the *depth* of the challenge tree, it can easily be adapted to prove CGKA security of TreeKEM, the main difference being the different challenge graph structure induced by blanking instead of tainting.

3.5 Security Proof for TTKEM in the ROM

The security of TTKEM is closely related to the notion of generalized selective decryption (GSD), which we adapt to the public key setting for our purposes:

Definition 7 (Generalized selective decryption (GSD), adapted from [20]). *Let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a public key encryption scheme with secret key space $\mathcal{K}$ and message space $\mathcal{M}$ such that $\mathcal{K} \subseteq \mathcal{M}$. The* GSD game *(for public key encryption schemes) is a two-party game between a challenger C and an adversary A. On input an integer N, for each $v \in [N]$ the challenger C picks a key pair $(\mathsf{pk}_v, \mathsf{sk}_v) \leftarrow \mathsf{Gen}(r)$ (where r is a random seed) 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:*

- *(**encrypt**, u, v): On input two nodes u and v, C returns an encryption $c = \mathsf{Enc}_{\mathsf{pk}_u}(\mathsf{sk}_v)$ of sk_v under pk_u along with pk_u and adds the directed edge (u, v) to $\mathcal{E}$. Each pair (u, v) can only be queried at most once.*
- *(**corrupt**, v): On input a node v, C returns sk_v and adds v to $\mathcal{C}$.*
- *(**challenge**, v), single access: On input a challenge node v, C samples $b \leftarrow \{0, 1\}$ uniformly at random and returns sk_v if $b = 0$, otherwise it returns a new secret key generated by Gen using a new independent uniformly random seed. In the context of GSD we denote the* challenge graph *as the graph induced by all nodes from which the challenge node v is reachable. We require that none of the nodes in the challenge graph are in $\mathcal{C}$, that $\mathcal{G}$ is acyclic and that the challenge node v is a sink. Note that A does not receive the public key of the challenge node, since it is a sink.*

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

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

We first prove a general result for our version of GSD, which could be of independent interest.

Theorem 3. *For any public key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and hash function H let the encryption scheme $\Pi' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ be defined as follows: 1) Gen' simply picks a random seed s as secret key and runs $\mathsf{Gen}(H(s))$ to obtain the corresponding public key, 2) Enc' is identical to Enc and 3) Dec', given the secret key s, extracts the secret key from $\mathsf{Gen}(H(s))$ and uses Dec to decrypt the ciphertext.*

If Π is $(\tilde{\epsilon}, t)$-IND-CPA secure and H is modelled as a random oracle, then Π' is (ϵ, t)-adaptive GSD secure, where $\epsilon = 2N^2 \cdot \tilde{\epsilon} + \frac{mN}{2^{\ell-1}}$, with N the number of nodes, m the number of oracle queries to H, ℓ the seed length.

Proof. Note that the GSD graph contains nodes $u \in [N]$ corresponding to seeds s_u that are hashed using the RO H to obtain seeds for the encryption scheme Π. We prove GSD security by a sequence of hybrids interpolating between the *real* game GSD_0 where the challenge query v is answered with the real seed s_v and the *random* game GSD_1 where it is answered with an independent uniformly random seed in S, where S denotes the seed space.

- Define $\mathsf{G}_0 := \mathsf{GSD}_0$, the real GSD game.
- Let $s' \in S$ and v be the challenge node. For $1 \leq i \leq \mathsf{indeg}(v)$ we define the hybrid game G_i as follows: The game is similar to G_{i-1} except that the ith query of the form $(\mathbf{encrypt}, u, v)$ is answered by $\mathsf{Enc}_{\mathsf{pk}_u}(s')$.

Note, the game $\mathsf{G}_{\mathsf{indeg}(v)}$ is distributed exactly the same as GSD_1. Thus, in this case, for any GSD-adversary A with advantage ϵ, the advantages of A in distinguishing hybrid games G_{i-1} from G_i sum up to at least ϵ. Since two subsequent hybrid games differ in exactly one encryption edge, we will use this distinguishing advantage to solve an IND-CPA challenge. To simulate game G_i, the reduction simply *guesses* the challenge node v as well as the source node u of the i th encryption incident on v. We denoote these guesses by v^* and u^*. This simulation, however, is only possible if A does not query its oracle H on any of the seeds corresponding to the parents of $v^* = v$, since otherwise A can trivially distinguish G_0 from $\mathsf{G}_{\mathsf{indeg}(v)}$. We encompass this issue by the following (more general) event.

- Event E: A queried a seed s^* to the random oracle which corresponds to a node that was not corrupted and is not reachable by any corrupted node, and no challenge query was issued for it.

Intuitively, event E is true if A queried the random oracle H on some seed which it doesn't trivially know and which is associated with a node that might end up in the challenge graph.

We make a case distinction on the probability $\gamma = \frac{\gamma_0+\gamma_1}{2}$, where $\gamma_b = \Pr[A \text{ triggers } E \mid \mathsf{GSD}_b]$ is the probability that A triggers E when playing the GSD_b game: If $\gamma < \epsilon/2$, apply Lemma 6, which results in an advantage $> \frac{\epsilon}{2N^2}$ against IND-CPA security. Else, apply Corollary 2 to obtain an IND-CPA adversary with advantage $> \frac{\epsilon}{2N^2} - \frac{m}{N2^\ell}$.

In either case we obtain an adversary with advantage $> \frac{\epsilon}{2N^2} - \frac{m}{N2^\ell}$ against IND-CPA security, which proves the claim.[18]

□

For ease of exposition, we prove a slightly weaker result in the main body of this paper and refer to the appendix for a more involved optimized proof. The proof of Lemma 4 – simplifying Lemma 6 in the appendix – relies on the piecewise guessing framework [17]. The framework allows for a more modular reduction and we believe that the proof of Lemma 4 already gives a good intuition on the final reduction presented in Lemma 6. This approach involves an additional loss in the indegree of the key graph. While the indegree is a small constant for the case of TTKEM, it can be up to linear in n for TreeKEM. Thus, the optimized proof of Theorem 3 is of particular interest if applied to TreeKEM, similar to Theorem 4. We mention that Panjwani [20] already gave a very similar proof for (private-key) GSD on graphs of bounded depth and the result we obtain in Lemma 6 is similar to Panjwani's result for the special case of graphs of depth 1. This is not a coincidence, since as long as the adversary does not query a seed in the challenge graph, the random oracle ensures that the adverdary is effectively playing a GSD game on a depth 1 graph. Since Panjwani's proof is very long and technical, we provide a self-contained proof in the appendix, which is precisely adapted to our setting of *public-key GSD in the random oracle model.* In the following, by the GSD game we refer to the public-key GSD game on the encryption scheme Π from Theorem 3.

We first consider the case of GSD adversaries which trigger event E only with small probability.

Lemma 4. *Let the event E be defined as in the proof of Theorem 3 and A an adversary against GSD such that*

- $\Pr[A \to 0 \mid \mathsf{GSD}_0] - \Pr[A \to 0 \mid \mathsf{GSD}_1] = \epsilon$ *and*

[18] Note, this reduction is *non-uniform.* One can get a uniform reduction by combining these two reductions and guessing whether event E will happen.

– $\gamma = \frac{\gamma_0+\gamma_1}{2} \leq \frac{\epsilon}{2}$ *where* $\gamma_b = \Pr[A \textit{ triggers } E \mid \mathsf{GSD}_b]$.

Then, in the ROM, there exists an IND-CPA adversary A' against the underlying encryption scheme with advantage $> \frac{\epsilon}{2\delta N^2}$, where δ is any upper bound on the maximium indegree of the key graph queried by A.

Proof. We first consider a variant GSD' of GSD where the game aborts as soon as E happens (we consider this as A outputting 1, i.e. no advantage can be gained on this event). We have

$$\begin{aligned}
\epsilon &= \Pr[A \to 0 \mid \mathsf{GSD}_0] - \Pr[A \to 0 \mid \mathsf{GSD}_1] \\
&= \Pr[(A \to 0) \wedge E \mid \mathsf{GSD}_0] - \Pr[(A \to 0) \wedge E \mid \mathsf{GSD}_1] \\
&\quad + \Pr[(A \to 0) \wedge \neg E \mid \mathsf{GSD}_0] - \Pr[(A \to 0) \wedge \neg E \mid \mathsf{GSD}_1] \\
&= \Pr[A \to 0 \mid E \wedge \mathsf{GSD}_0] \cdot \gamma_0 - \Pr[A \to 0 \mid E \wedge \mathsf{GSD}_1] \cdot \gamma_1 \\
&\quad + \Pr[(A \to 0) \wedge \neg E \mid \mathsf{GSD}_0] - \Pr[(A \to 0) \wedge \neg E \mid \mathsf{GSD}_1] \\
&\leq \frac{\epsilon}{2} + \Pr[(A \to 0) \wedge \neg E \mid \mathsf{GSD}_0] - \Pr[(A \to 0) \wedge \neg E \mid \mathsf{GSD}_1]
\end{aligned}$$

So clearly A has advantage $> \epsilon/2$ in GSD'. We now show that GSD'_0 and GSD'_1 are indistinguishable. We will do so by applying the piecewise guessing technique from Theorem 1. To this aim, first consider the *selective* version of the GSD' game, where the adversary has to commit to all its queries in the beginning of the game, i.e., the entire graph structure, the challenge v, as well as the subset of corrupted nodes are known to the reduction before executing the GSD game. Let H_i denote the selective version (with $v^* = v$) of the game G_i for $i \in [\mathsf{indeg}(v^*)]$.

From an adversary that distinguishes two subsequent games H_{i-1} and H_i one can easily construct an adversary A' with the same advantage against the IND-CPA game: A' chooses two random seeds s, s' as its messages and receives a challenge ciphertext c in return. It then simulates the GSD' game to A and embeds the challenge c in the edge where H_{i-1} and H_i differ. Denote this edge by (u, v). Since A' generates almost all of the remaining information of the hybrid itself, it can respond faithfully to any query of the adversary. The only tricky queries are 1) a corruption of u and 2) encryption queries of the form (i, u), since this requires to respond with (an encryption of) a seed s_u such that $H(s_u) = r_u$ and applying the key generation algorithm to r_u results in the public key associated to u. Luckily, since u is a parent of the challenge v, u may not be corrupted nor reachable from a corrupted node, so querying H for s_u would trigger E and the game aborts. It follows that we can assign a random seed to u, which to A is information-theoretically indistinguishable to using the correct seed. Now clearly A' has the same advantage as A since if c encrypts s it is simulating H_{i-1}, while otherwise A' is simulating H_i.

In the adaptive setting, A' can simulate game G_i by only guessing the challenge node (v) and the source (u) of the ith edge incident on v. Thus, it requires $2\log(N)$ bits of information on the adversary's choices. Then by Theorem 1 it follows that the advantage of A' is at least $\frac{\epsilon}{\delta \cdot N^2}$.

□

The following Lemma shows that any GSD adversary triggering event E can be reduced to an adversary against a partially selective version of GSD where the challenger aborts whenever event E happens.

Lemma 5. *Let the event E be defined as in the proof of Theorem 3. We consider a slightly modified version of GSD: let GSD''_b be defined like GSD_b with the two differences that a) the game will abort if E happens (we will consider this equivalent to the adversary outputting 1) and b) the adversary has to commit to the challenge node at the beginning of the game, i.e. GSD'' is partially selective. Then there exists an adversary A in the GSD'' game on $N+1$ nodes such that for any adversary A^- which triggers event E with probability ϵ in the GSD game on N nodes, we have*

$$\Pr[A \to 0 \mid \mathsf{GSD}''_0] - \Pr[A \to 0 \mid \mathsf{GSD}''_1] \geq \frac{\epsilon}{N} - \frac{m}{2^\ell}.$$

Proof. In the following we consider event E with respect to A^- playing the normal GSD game. We construct the adversary A as follows. First, A *guesses* the node v^* which will be associated with the seed which turns

E true, i.e., A samples $v^* \leftarrow [N]$ uniformly at random and issues a challenge query for v^*; let s^* be the response it receives. Then it runs A^-. If A^- queries $(\mathbf{encrypt}, u, u')$ for $u \neq v^*$, then A just forwards this query to the challenger and returns to A^- whatever it receives in response. For $u = v^*$, on the other hand, A issues a query $(\mathbf{encrypt}, N+1, u')$. Note that the node $N+1$ is never used by A^- and the adversary will associate the key pk_{N+1} it receives to the node v^*. Unless a seed associated with v^* is queried to the random oracle H, this is indistinguishable from the real GSD game and if this event happens then – assuming the guess v^* was correct – this means that event E turned true. Since A aims to use the event E to break its own GSD challenge, it is enough that the game simulated to A^- until E happens is indistinguishable from the real GSD game.

If A^- queries $(\mathbf{corrupt}, u)$, A simply forwards this query to the challenger and returns the response it receives. If A^- issues a corruption or encryption query such that v^* is reachable from a corrupted node, A aborts and outputs 1.

When A^- queries $(\mathbf{challenge}, v)$ for $v \neq v^*$, A sends $(\mathbf{corrupt}, v)$ to the challenger and receives s_v. Then it samples a bit $c \leftarrow \{0,1\}$ uniformly at random, sets $s_0 := s_v$, samples $s_1 \leftarrow S$ unifomly at random, and returns s_c to A^-. Note that v must be a sink node, hence this corruption does not affect the validity of v^* as a GSD challenge. If A^- happens to choose v^* as its challenge, A aborts and outputs 1.

If A^- makes an oracle query $H(s)$, then A just forwards this query to the challenger if $s \neq s^*$, otherwise A aborts and outputs the bit $b = 0$. If the seed s^* is never queried (and the game GSD" does not abort), A outputs 1.

Let V denote the event that v^* is associated with the seed which turns E true. Let b^* be the GSD" challenge bit. Now we first note that

$$\Pr[A \to 0 \mid b^* = 1] \leq \frac{m}{2^\ell}$$

because if $b^* = 1$, s^* is information-theoretically hidden from A^-. Furthermore,

$$\begin{aligned}
\Pr[A \to 0 \mid b^* = 0] &= \Pr[(A \to 0) \wedge E \mid b^* = 0] && \text{because } A \to 0 \text{ implies } E \\
&= \epsilon \cdot \Pr[A \to 0 \mid E \wedge (b^* = 0)] && \text{because } \Pr[E \mid b^* = 0] = \Pr[E] = \epsilon \\
&= \epsilon \cdot \Pr[(A \to 0) \wedge V \mid E \wedge (b^* = 0)] && \text{since } \neg V \text{ implies } A \to 1 \\
&= \epsilon \cdot \Pr[A \to 0 \mid V \wedge E \wedge (b^* = 0)] \Pr[V \mid E \wedge (b^* = 0)] \\
&\geq \frac{\epsilon}{N}
\end{aligned}$$

Where the last step follows since v^* was chosen uniformly at random and the simulation is independent of this choice until E happens and/or A aborts, and because $V \wedge E \wedge (b^* = 0)$ implies $A \to 0$. In summary, we obtain

$$\Pr[A \to 0 \mid b^* = 0] - \Pr[A \to 0 \mid b^* = 1] \geq \frac{\epsilon}{N} - \frac{m}{2^\ell}.$$

This proves the claim. □

The following Corollary now gives a reduction for GSD adversaries which trigger event E with large probability. We optimize the result in Corollary 2 in the appendix.

Corollary 1. *Let the event E be defined as in the proof of Theorem 3, and A an arbitrary GSD adversary which triggers E with probability ϵ. Then there exists an IND-CPA adversary A' with advantage*

$$\Pr[A' \to 0 \mid b^* = 0] - \Pr[A' \to 0 \mid b^* = 1] \geq \frac{\epsilon}{\delta(N^2+N)} - \frac{m}{\delta(N+1)2^\ell}.$$

where δ is any upper bound on the maximium indegree of the graph queried by A.

Proof. We first apply Lemma 5 to construct an adversary against GSD" with advantage $\frac{\epsilon}{N} - \frac{m}{2^\ell}$. Then we apply a very similar proof as for Lemma 4 to construct an IND-CPA adversary: This proof is exactly the same with the only difference being that GSD" (played on $N+1$ nodes) is less adaptive, i.e. A commits to the challenge node v at the beginning of the game, so when applying Theorem 1 to switch from the selective to the adaptive version, A' does not need to guess it. It follows that in order to simulate game G_i, A' needs to guess only the source u of the ith encryption edge incident on v, which consists of $\log N$ many bits. Accordingly, by Theorem 1, the IND-CPA adversary has advantage

$$\frac{1}{\delta(N+1)}\left(\frac{\epsilon}{N}-\frac{m}{2^{\ell}}\right).$$

□

We now adapt the above proof to show a polynomial time reduction for TTKEM in the random oracle model. Intuitively, the CGKA graph corresponds to a GSD graph in the above sense (i.e. for the transformed Π', where H_2 plays the role of the RO), with the only difference that there are additional edges corresponding to a second RO H_1. The following Theorem shows that this difference does not impact security.

Theorem 4. *If the encryption scheme used in TTKEM is $(\tilde{\epsilon}, t)$-IND-CPA secure and H_1, H_2 are modelled as random oracles, then TTKEM is (Q, ϵ, t)-CGKA-secure, where $\epsilon = \tilde{\epsilon} \cdot 8(nQ)^2 + \mathsf{negl}$.*

Proof (Sketch). In order to adapt the proof of Theorem 3 to the setting where some seeds may be derived from others by the random oracle H_1, we first slightly change the event E to also include queries to H_1, i.e. the event E is now that A queries H_1 *or* H_2 on a seed that it doesn't trivially know through corruptions or the challenge query. The case distinction in the main proof on the probability of A remains the same.

The case where E happens with relatively small probability $< \epsilon/2$ (cf. Lemma 4) is easily handled, since in this case the reduction may generate all seeds independently as before and when nodes get corrupted, it can simply program H_1 to ensure consistency. The seeds in the challenge graph are not consistent in this simulation, but since the adversary does not query any of these seeds with high (enough) probability, this is indistinguishable and our reduction retains most of the advantage.

The other case, where E happens with large probability $\geq \epsilon/2$ is handled similarly. The key observation in Lemma 5 is that it is sufficient to simulate the GSD game correctly until E happens. Using the approach of programming H_1, this is still the case and thus Corollary 1 also holds in this case.

We conclude by observing that the CGKA graph has at most $N = 2nQ$ nodes and that none of the seeds and secret keys in the challenge graph (for any safe group key) is leaked by Lemma 2. □

We remark that one can easily adapt the proof to the case of TreeKEM (with blanking) since the loss in security only depends on the maximal number of nodes in the challenge graph but not on its structure.

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, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, 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. M. Bellare, A. C. Singh, J. Jaeger, M. Nyayapati, and I. Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In J. Katz and H. Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 619–650. Springer, Heidelberg, Aug. 2017.
5. C. H. Bennett. Time/space trade-offs for reversible computation. *SIAM J. Comput.*, 18(4):766–776, 1989.
6. K. Bhargavan, R. Barnes, and E. Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups. May 2018.
7. R. Canetti, J. A. Garay, G. Itkis, D. Micciancio, M. Naor, and B. Pinkas. Multicast security: A taxonomy and some efficient constructions. In *IEEE INFOCOM'99*, pages 708–716, New York, NY, USA, Mar. 21–25, 1999.
8. K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, and K. Milner. On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees. CCS, 2018.
9. C. Cremers, B. Hale, and K. Kohbrok. Efficient post-compromise security beyond one group. Cryptology ePrint Archive, Report 2019/477, 2019. `https://eprint.iacr.org/2019/477`.
10. F. B. Durak and S. Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In N. Attrapadung and T. Yagi, editors, *IWSEC 19*, volume 11689 of *LNCS*, pages 343–362. Springer, Heidelberg, Aug. 2019.
11. A. Fiat and M. Naor. Broadcast encryption. In D. R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 480–491. Springer, Heidelberg, Aug. 1994.
12. 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, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 601–620. Springer, Heidelberg, Aug. 2015.
13. G. Fuchsbauer, C. Kamath, K. Klein, and K. Pietrzak. Adaptively secure proxy re-encryption. In D. Lin and K. Sako, editors, *PKC 2019, Part II*, volume 11443 of *LNCS*, pages 317–346. Springer, Heidelberg, Apr. 2019.
14. X. Gabaix. Zipf's law for cities: An explanation. *The Quarterly Journal of Economics*, 114(3):739–7675, 1999.
15. IETF. The Messaging Layer Security (MLS) Protocol (draft-ietf-mls-protocol-09). https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/, March 2020.
16. J. Jaeger and I. Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 33–62. Springer, Heidelberg, Aug. 2018.
17. Z. Jafargholi, C. Kamath, K. Klein, I. Komargodski, K. Pietrzak, and D. Wichs. Be adaptive, avoid overcommitting. In J. Katz and H. Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 133–163. Springer, Heidelberg, Aug. 2017.
18. D. Jost, U. Maurer, and M. Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Y. Ishai and V. Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 159–188. Springer, Heidelberg, May 2019.
19. Matthew A. Weidner. Group Messaging for Secure Asynchronous Collaboration. Master's thesis, University of Cambridge, June 2019.
20. S. Panjwani. Tackling adaptive corruptions in multicast encryption protocols. In S. P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 21–40. Springer, Heidelberg, Feb. 2007.
21. T. Perrin and M. Marlinspike. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/, 2016.
22. B. Poettering and P. Rösler. Towards bidirectional ratcheted key exchange. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 3–32. Springer, Heidelberg, Aug. 2018.
23. D. Rybski, S. V. Buldyrev, S. Havlin, F. Liljeros, and H. A. Makse. Scaling laws of human interaction activity. *Proceedings of the National Academy of Sciences*, 106(31):12640–12645, 2009.
24. D. M. Wallner, E. J. Harder, and R. C. Agee. Key management for multicast: Issues and architectures. Internet Draft, Sept. 1998. `http://www.ietf.org/ID.html`.
25. C. K. Wong, M. G. Gouda, and S. S. Lam. Secure group communications using key graphs. In *Proceedings of ACM SIGCOMM*, pages 68–79, Vancouver, BC, Canada, Aug. 31 – Sept. 4, 1998.

A Pseudocode for TTKEM specification

A.1 Notation

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. Further, 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'. We will need to sample fresh random seeds to generate new key pairs when refreshing a path, we do this through `gen-seed`.

A.2 Path partitions

When updating, a user needs to partition the set of extra nodes to be refreshed (nodes not on their path with a tainted ancestor) into paths, so that a single seed can be used to update each path. Formally, for a user id, we want a set of paths $P_i = \{v_{i,0}, \dots, v_{i,m_i}\}$ such that every tainted node is in some path P_i and moreover:

- `child`$(v_{i,j}) = v_{i,j+1}$ for $j < m_i$ *(P_i is a path)*
- $v_{i,j} \neq v_{k,l}$ if $i \neq k$ for any j, l *(each node is only in one path)*
- `get_tainter`$(v_{i,0}) = id$ *(the start of each path is a node tainted by id)*
- $\forall i, j$: `child`$(v_{i,m_i}) \neq v_{j,0}$ *(paths are maximal)*
- $P_i \bigcap P_{id} = \emptyset$ *(paths are disjoint from main path to root)*
- `child`$(v_{i,m_i}) \in P_{id} \vee$ `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 in a graphical representation of the tree (any total ordering on vertices suffices). We denote this ordered partition by `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 `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 $i < j$.

A.3 TTKEM Dynamics in detail

In this section we provide a more detailed description of the group operations together with pseudo-code for them.

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}. We use the pointer `me` to refer to the identity of the user sending the protocol message.

```
refresh ($\gamma, \mathsf{ID}, T$)
    $P_0 \leftarrow \gamma.\mathtt{path}(\mathsf{ID})$
    $\{P_1, \dots, P_n\} \leftarrow \gamma.\mathtt{tainted\text{-}by}(\mathsf{ID})$ #refresh all paths from tainted nodes to root
    for $i = n, \dots, 0$ do
        $v_{i,0}, \dots, v_{i,m} \leftarrow P_i$
        $\{\Delta_{i,0}, \dots \Delta_{i,m}\} \leftarrow \mathtt{expand}(\mathtt{gen\text{-}seed}(), m+1)$
        for $p \in \mathit{parents}(v_{i,0})$ do
            #encrypt first to parents of 1st node
            if $p \neq \perp$ then
                $T_{new\_seeds}.insert(\mathsf{Enc}_{\gamma.\mathtt{get\_pk}(p)}(\Delta_{i,0}))$
        refresh-node($\gamma, v_{i,0}, \Delta_{i,0}, T$)
        for $j = 1, \dots, m$ do
            $T_{new\_seeds}.insert(\mathsf{Enc}_{\gamma.\mathtt{get\_pk}(\gamma.\mathtt{partner}(v_{i,j-1}))}(\Delta_{i,j}))$
            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 ($\gamma, v, \Delta, T$)
    if $v = v_{root}$ then
        $\gamma.\mathtt{set\_sk}(v_{root}, \Delta)$
    else
        $(sk, pk) \leftarrow \mathsf{Gen}(H_2(\Delta))$
        $\gamma.\mathtt{set\_pk}(v, pk)$;
        $\gamma.\mathtt{set\_tainter}(v, \mathtt{me})$
        $T_{new\_pks}.insert(pk)$
        if $v \in \gamma.\mathit{path}(\mathsf{ID})$ then
            $\gamma.\mathtt{set\_sk}(v, sk)$
```

Initialize. To create a new group with parties $\{\mathsf{ID}_1, \dots, \mathsf{ID}_n\}$, 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 either knowledge of Alice's secret key or control/knowledge of the randomness used. We use h to refer to the hash function used.

```
add $(\gamma, \mathsf{ID}, pk)$
    $\gamma' \leftarrow \gamma$
    $\gamma'.\texttt{add\_party}(\mathsf{ID}, pk)$
    $\{v_0, \ldots, v_d\} \leftarrow \gamma'.\texttt{path}(\mathsf{ID})$
    $sk \leftarrow \gamma'.\texttt{get\_sk}(\gamma'.\texttt{index}(\texttt{me}))$
    $r \leftarrow \$; \Delta \leftarrow h(sk, r)$
    $\{\Delta_0, \ldots, \Delta_d\} \leftarrow \texttt{expand}(\Delta, d+1)$
    $\textit{refresh-node}(\gamma', v_0, \Delta_0, T)$
    for $i = 1, \ldots, d$ do
        $u \leftarrow \gamma.\texttt{partner}(v_{i-1})$
        if $u \neq \perp$ then
            $T_{new\_seeds}.insert(\mathsf{Enc}_{\gamma.\texttt{get\_pk}(u)} \Delta_i)$
        $\textit{refresh-node}(\gamma, v_i, \Delta_i, T)$
    $T_{op} \leftarrow (add, \mathsf{ID}, pk)$
    $T_{sender} \leftarrow \texttt{me}$
    $T_{\mathcal{H}} \leftarrow \gamma.\texttt{get\_hash}()$
    $\gamma'.\texttt{update\_hash}(T)$
    $W_{op} \leftarrow welcome$
    $W_{sender} \leftarrow \texttt{me}$
    $W_{seed} \leftarrow \mathsf{Enc}_{pk}(\Delta)$
    $W_{\mathcal{T}} \leftarrow \gamma'.\texttt{get\_tree}()$
    $W_{\mathcal{H}} \leftarrow \gamma.\texttt{get\_hash}()$
    $W_{\mathcal{M}} \leftarrow \gamma.\texttt{get\_members}()$
    return$(\gamma', W, T)$
```

(a)

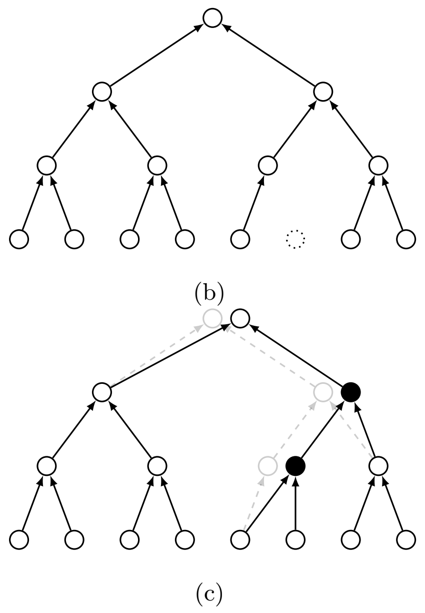
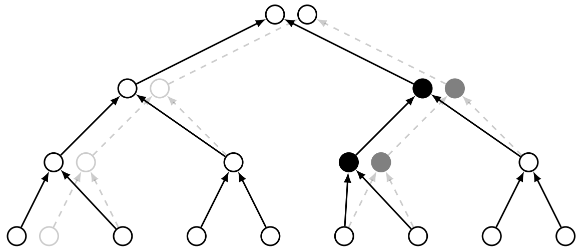


Fig. 10: (a) Pseudocode for the Add operation. Sample Add operation: (b) illustrates the state of the tree before Alice adds Frank (6th node) after which it turns into (c).

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`.

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

(a)



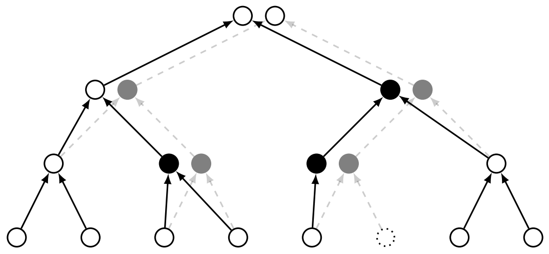
(b)

Fig. 11: (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.

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.

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

(a)



(b)

Fig. 12: (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 get_enc, *update-path* and *proc-refresh* as subroutines. The function get_enc 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. Finally, *proc-refresh* takes a user (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 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**
            $(sk, \_) \leftarrow \mathsf{Gen}(H_2(\Delta))$
            $\Delta \leftarrow H_1(\Delta)$
            $\gamma.\texttt{set\_sk}(v, sk)$

**proc-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**
        $\{v_0, \ldots, v_n\} \leftarrow \texttt{intersection}(P_i, \gamma.\texttt{path}(\texttt{me}))$
        $enc \leftarrow \mathsf{get\_enc}(\gamma, v_0, P_0, T_{new\_seeds})$
        $(p_l, p_r) \leftarrow \gamma.\texttt{parents}(v_0)$
        **if** $p_l \neq \perp \wedge p_l \in \gamma.\mathit{path}(\mathit{me})$ **then**
            $sk \leftarrow \gamma.\texttt{get\_sk}(p_l)$
        **else**
            $sk \leftarrow \gamma.\texttt{get\_sk}(p_r)$
        $update\text{-}path(\gamma, \{v_0, \ldots, v_n\}, \mathsf{Dec}_{sk}(enc), sender)$
```

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** $(\gamma, T)$
    **req** $T_{\mathcal{H}} = \gamma.\texttt{get\_hash}()$
    **if** $T_{op} = upd$ **then**
        $proc\text{-}refresh(\gamma, T_{new\_keys}, T_{sender}, T_{sender})$
        $\gamma.\texttt{update\_pks\_and\_tainter}(T_{new\_pks}, T_{sender}, T_{sender})$
    **if** $T_{op} = (rem, \mathsf{ID})$ **then**
        **if** $\mathsf{ID} \neq \mathit{me}$ **then**
            $proc\text{-}refresh(\gamma, T_{new\_keys}, \mathsf{ID}, T_{sender})$
            $\gamma.\texttt{update\_pks\_and\_tainter}(T_{new\_pks}, \mathsf{ID}, T_{sender})$
            $\gamma.\texttt{remove\_party}(\mathsf{ID})$
        **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**
        $\gamma.\texttt{add\_party}(\mathsf{ID}, pk)$
        $proc\text{-}refresh(\gamma, T_{new\_keys}, \mathsf{ID}, T_{sender})$
        $\gamma.\texttt{update\_pks\_and\_tainter}(T_{new\_pks}, \mathsf{ID}, T_{sender})$
    **if** $T_{op} = welcome$ **then**
        $\gamma.\texttt{init\_state}(T_{\mathcal{M}}, T_{\mathcal{T}}, T_{\mathcal{H}})$
        $update\text{-}path(\gamma, \{\gamma.\texttt{index}(\texttt{me}), \ldots, v_{root}\}, \mathsf{Dec}_{sk}(T_{seed}), T_{sender})$
    **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**$(\gamma, \mathsf{key}(\gamma))$
```

B Proof of Lemma 1

Lemma 1 *In TreeKEM, the number of ciphertexts required to transition from a completely blank tree (except for the creator's path) to a full tree, without blanks, is at least* $n(\log n - 2) + 2$.

Proof. 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:

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

To see this, let $\mathcal{T}_1$ and $\mathcal{T}_\in$ denote the left and right subtrees respectively. If any user (with a leaf) in $\mathcal{T}_1$ updates before anyone in $\mathcal{T}_2$ does, $\mathcal{T}_2$ will be blank. Hence, communicating the new group key to users in $\mathcal{T}_\in$ will require one ciphertext per user. On the contrary, if some update from $\mathcal{T}_2$ has already taken place, all updates from $\mathcal{T}_1$ will just require a single ciphertext to be communicated to $\mathcal{T}_2$: an encryption of 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 from a user in $\mathcal{T}_\infty$ should come first from the right subtree of $\mathcal{T}_1$, assume w.l.o.g. it comes from the left-most user in that subtree, and so on (similarly for $\mathcal{T}_2$). Thus, we get that the sequence above is optimal.

Now, the cost (i.e. number of required ciphertexts) to update in this order is $(n/2-1)+1$ for the first update, $(n/4-1)+2$ 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:

$$\begin{aligned}
&\frac{n}{2} + 2\left(\frac{n}{4}+1\right) + 4\left(\frac{n}{8}+2\right) + \ldots + \frac{n}{4}\left(\frac{n}{n/2} + \log n - 2\right) = \\
&= \frac{n}{2}(\log n - 1) + \sum_{i=1}^{\log n - 1} (i-1)2^{i-1} \\
&= \frac{n}{2}(\log n - 1) + 2\left(2^{\log n - 2}(\log n - 2) - 2^{\log n - 2} + 1\right) \\
&= n(\log n - 2) + 2
\end{aligned}$$

□

C Optimize Lemma 4 and Corollary 1

In the proof of Lemma 4 we considered a variant GSD' of GSD, where the challenger aborts once event E is triggered. Here we optimize the result by prooving that GSD'-security reduces to IND-CPA security of the underlying encryption scheme at a loss in security which is *independent* of the degree of the key graph.

Lemma 6. *Let the event E be defined as in the proof of Theorem 3 and A an adversary against GSD' with advantage ϵ, where GSD' is defined similar to GSD but the challenger aborts once event E is triggered. Then, in the ROM, there exists an IND-CPA adversary A' against the underlying encryption scheme with advantage $> \frac{\epsilon}{N^2}$.*

Proof. We define an adversary A' against IND-CPA security of Π as follows (see Figure 13): On receipt of a challenge public key pk^*, A' samples two independent uniformly random seeds $s, s' \in S$ and sends them to the IND-CPA challenger. In response it receives a challenge ciphertext c^* which encrypts either s in case $b^* = 0$ or s' if $b^* = 1$. To guess the secret bit b^*, the algorithm A' now attempts to simulate a hybrid game to A and embed the challenge c^* in such a way that, for some i, the execution of the game will look just like G'_{i-1} if $b^* = 0$ and like G'_i (where G'_i is a variant of G_i from the proof of Theorem 3 where the game aborts in case E happens). To this aim, A' first samples $v^* \leftarrow [N]$ uniformly at random (i.e., it guesses the challenge node). Then it samples independent seeds $s_u \in S$ uniformly at random for all nodes $u \in [N] \setminus \{v^*\}$ and sets $s_{v^*} := s$. Instead of sampling a fresh seed s', A' just uses the seed s' which it used in the IND-CPA game. Hence, both s_{v^*} and s' are associated with node v^*. Next, A' chooses $u^* \leftarrow [N]$ uniformly at random and for all $u \in [N] \setminus \{u^*\}$ queries the random oracle H on s_u to receive $r_u := H(s_u)$. Just the same as in the GSD game, A' then computes $(\mathsf{pk}_u, \mathsf{sk}_u) \leftarrow \mathsf{Gen}(r_u)$. To embed the challenge, it sets $\mathsf{pk}_{u^*} := \mathsf{pk}^*$.

A' now answers all encryption queries just as in the game GSD_0 except for the edges incident on v^*: All queries $(\textbf{encrypt}, u, v^*)$ are answered with encryptions $\mathsf{Enc}_{\mathsf{pk}_u}(s')$, until A queries $(\textbf{encrypt}, u^*, v^*)$, which is answered with c^*; all further queries are answered with encryptions of s. All corrupt queries are answered just the same as in the GSD game and the challenge query v is answered with s_v. For all queries A makes to the random oracle, A' just queries its own oracle H and returns the output.

In this way A' aims to simulate some game G'_i and this simulation is perfect except for the case that A queries the random oracle H on s_{u^*}: To answer this query consistently, A' would have to output a seed r_{u^*} satisfying $\mathsf{Gen}(r_{u^*}) = (\mathsf{pk}^*, \mathsf{sk}^*)$. This however implies that either 1) the node u^* was challenged or corrupted, hence, u^* can not be a parent of the challenge node so the simulation failed due to an incorrect guess anyway; or 2) A triggered event E, in which case G'_i aborts and thus A' can simply abort and output 1, which faithfully preserves the simulation. Accordingly, as soon as A' sees that its guesses u^* and v^* are incorrect, i.e., v^* is not the challenge node or u^* is corrupted or reachable from a corrupted node, it aborts the game and outputs $b' = 1$.

The following events will now determine the output behaviour of A':

- Event U: A queried $(\textbf{encrypt}, u^*, v)$.
- Event V: The challenge node v coincides with v^*.

After running the game with A, the reduction A' evaluates whether these events happened and chooses its final output b' as follows: If U and V are true, this means that A' correctly guessed the challenge node v, managed to embed the IND-CPA challenge key pk^* at u^* and some game G'_i was executed. In this case A' outputs the bit b it received from A. Note that if $(\textbf{encrypt}, u^*, v^*)$ happens to be the first encryption query incident on the challenge node $v = v^*$ and $b^* = 0$, then the game simulated to A looks exactly the same as GSD'_0. If $(\textbf{encrypt}, u^*, v^*)$ happens to be the last encryption query incident on the challenge node $v = v^*$ and $b^* = 1$, then the game simulated to A looks exactly the same as GSD'_1. If either U or V fails, as mentioned above, A' outputs $b' = 1$.

If A either doesn't make a challenge query or the challenge node v is a source node, then the advantage $\tilde{\epsilon}$ of A in winning the GSD game is 0 (information-theoretically; note by definition the challenge node must be a sink). Hence, we assume that v is not a source node. Furthermore, we assume that v constitutes a valid GSD challenge, since otherwise, again, the advantage of A is 0. If any of these assumptions fail, A' outputs $b' = 1$.

For the advantage of A' in breaking the IND-CPA security of the encryption scheme we have

$$\Pr[A' \to 0 \mid b^* = 0] - \Pr[A' \to 0 \mid b^* = 1] = \Pr[(A' \to 0) \wedge U \wedge V \mid b^* = 0] - \Pr[(A' \to 0) \wedge U \wedge V \mid b^* = 1]$$

For technical reasons, we slightly modify A' as follows: After the interaction with A, the reduction A' one by one adds further edges incident on v, where it starts with edges outgoing from nodes which are neither corrupted nor reachable from a corrupted node, if such nodes exist. This neither changes the output behaviour of A' nor validity of the challenge node, since it happens after the execution of the GSD' game. We consider $N-1$ disjoint events U_i denoting the event that u^* is the source of the ith encryption edge incident on v (with respect to the chronological order of the queries). Let δ be an upper bound on the indegree of the (final) key graph $\mathcal{G}$. We denote the event that A queries exactly j encryptions incident on the challenge node by Δ_j. Hence, if V, U_i and Δ_j for some $j \geq i$ happen, then depending on the challenge bit b^*, the game simulated to A is distributed exactly the same as G'_{i-1} or G'_i. Since V and $A' \to 0$ implies U and we

```
$\mathcal{B}^{C,\mathcal{A},H}_{\Pi=(\mathsf{Gen},\mathsf{Enc},\mathsf{Dec})}(\mathsf{pk}^*)$
$\hat{U}, V, E, \mathsf{FAIL} \leftarrow$ false
$\mathcal{E}, \mathcal{C} \leftarrow \emptyset$
$\mathcal{G} \leftarrow ([N], \mathcal{E})$
$s^* \leftarrow \perp$
$v^* \leftarrow_\$ [N]$
$s, s' \leftarrow_\$ S$
$(\mathsf{pk}^*, c^*) \leftarrow_\$ C(s, s')$
for $u \in [N] \setminus \{v^*\}$
  $s_u \leftarrow_\$ S$
$s_{v^*} \leftarrow s$
$u^* \leftarrow_\$ [N]$
for $u \in [N] \setminus \{u^*\}$
  $r_u \leftarrow H(s_u)$
  $(\mathsf{pk}_u, \mathsf{sk}_u) \leftarrow_\$ \mathsf{Gen}(r_u)$
$\mathsf{pk}_{u^*} \leftarrow \mathsf{pk}^*$
$b \leftarrow \mathcal{A}^{\mathcal{B}\text{-enc},\mathcal{B}\text{-corr},\mathcal{B}\text{-chall},\mathcal{B}\text{-}H}$
if no challenge query was issued
  or $v$ reachable in $\mathcal{G}$ from a node in $\mathcal{C}$
  or $v$ not a sink node in $\mathcal{G}$
  or $v$ a source node in $\mathcal{G}$
    $\mathsf{FAIL} \leftarrow$ true
if $\neg V$ or $\neg\hat{U}$ or $E$ or $\mathsf{FAIL}$
  return 1
else
    return $b$
```

```
$\mathcal{B}$-enc$(u_1, u_2)$
$\mathcal{E} \leftarrow \mathcal{E} \cup \{(u_1, u_2)\}$
if $u_2 = v^*$
  if $u_1 = u^*$
    $\hat{U} \leftarrow$ true
    return $c^*$
  else if $\hat{U}$
    return $\mathsf{Enc}_{\mathsf{pk}_{u_1}}(s)$
  else
    return $\mathsf{Enc}_{\mathsf{pk}_{u_1}}(s')$
return $\mathsf{Enc}_{\mathsf{pk}_{u_1}}(s_{u_2})$
```

```
$\mathcal{B}$-corr$(u)$
$\mathcal{C} \leftarrow \mathcal{C} \cup \{u\}$
return $s_u$
```

```
$\mathcal{B}$-chall$(v)$
// one time use only
if $v = v^*$
  $V \leftarrow$ true
$\mathcal{C} \leftarrow \mathcal{C} \cup \{v\}$
return $s_v$
```

```
$\mathcal{B}$-$H(\bar{s})$
if $\bar{s} = s_{u^*}$
  FAIL $\leftarrow$ true
  return $\perp$
if ($\bar{s} = s_u$ for some $u \in [N] \setminus \{v\}$)
      or (not $V$ and $\bar{s} = s'$)
  if $u$ (resp. $v$) is not reachable in $\mathcal{G}$
        from any of the $u' \in \mathcal{C}$
    $E \leftarrow$ true
    abort $\mathcal{A}$
return $H(\bar{s})$
```

Fig. 13: Security Reduction for RO proof. C is the IND-CPA challenger.

are guaranteed that one of the U_i and one of the Δ_j happens, we have for all i, j:

$$\Pr[(A' \to 0) \wedge U \wedge V \mid b^* = 0]$$
$$= \sum_{j=1}^{\delta} \sum_{i=1}^{N-1} \mathsf{Pr}[A' \to 0 \mid \Delta_j \wedge V \wedge U_i \wedge (b^* = 0)] \cdot \mathsf{Pr}[\Delta_j \mid V \wedge U_i \wedge (b^* = 0)] \cdot \mathsf{Pr}[V \wedge U_i \mid b^* = 0] \quad (1)$$

and similarly for the case $b^* = 1$. Let U^* denote the subevent of U that u^* is neither corrupted nor reachable from a corrupted node. Then for all $i \in [2, N-1]$, (since E doesn't happen) the game simulated to A is identically distributed in the two cases $U^* \wedge U_i \wedge (b^* = 0)$ and $U^* \wedge U_{i-1} \wedge (b^* = 1)$. This is true no matter if the ith encryption was queried by A or not. Thus, we have

$$\mathsf{Pr}[A' \to 0 \mid \varDelta_j \wedge V \wedge U^* \wedge U_i \wedge (b^* = 0)] = \mathsf{Pr}[A' \to 0 \mid \varDelta_j \wedge V \wedge U^* \wedge U_{i-1} \wedge (b^* = 1)].$$

On the other hand, if U^* is false and u^* is corrupted or reachable by a corrupted node, then A' outputs 1, independently of any other events to happen. For the probability that U^* happens, note that if A makes at least i encryption queries incident on the challenge node v, then U^* must be true in both cases $V \wedge U_i \wedge (b^* = 0)$ and $V \wedge U_{i-1} \wedge (b^* = 1)$. Thus, we obtain for all $j \in [\delta]$, $i \in [2, N-1]$ such that $j \geq i$:

$$\begin{aligned}\mathsf{Pr}[A' \to 0 \mid & \varDelta_j \wedge V \wedge U_i \wedge (b^* = 0)] \\ &= \mathsf{Pr}[A' \to 0 \mid \varDelta_j \wedge V \wedge U^* \wedge U_i \wedge (b^* = 0)] \cdot \mathsf{Pr}[U^* \mid \varDelta_j \wedge V \wedge U_i \wedge (b^* = 0)] \\ &= \mathsf{Pr}[A' \to 0 \mid \varDelta_j \wedge V \wedge U^* \wedge U_{i-1} \wedge (b^* = 1)] \cdot \mathsf{Pr}[U^* \mid \varDelta_j \wedge V \wedge U_{i-1} \wedge (b^* = 1)] \\ &= \mathsf{Pr}[A' \to 0 \mid \varDelta_j \wedge V \wedge U_{i-1} \wedge (b^* = 1)].\end{aligned} \tag{2}$$

If A queries less than i encryption queries incident on v, then U_i implies that A' outputs 1, independently of b^*. Thus, for all $j \in [\delta]$, $i \in [2, N-1]$ such that $j < i$:

$$\mathsf{Pr}[A' \to 0 \mid \varDelta_j \wedge V \wedge U_i \wedge (b^* = b)] = 0 \tag{3}$$

for either $b \in \{0, 1\}$. By a similar argument as above, we have for all $j > i$:

$$\Pr[\varDelta_j \mid V \wedge U_i \wedge (b^* = 0)] = \Pr[\varDelta_j \mid V \wedge U_{i-1} \wedge (b^* = 1)] \tag{4}$$

In fact, (4) also holds in case $j = i$, but this is more subtle. Here, it is crucial that A' starts with uncorrupted nodes when choosing additional edges after the interaction with A: Clearly, if there are only i nodes (including v) which are neither corrupted nor reachable from a corrupted node, then $\varDelta_i$ cannot happen. On the other hand, if there are more than i such nodes, then U_i implies that u^* is such a node and the claim follows by noting that after the $i-1$th query the case $V \wedge U_{i-1} \wedge (b^* = 1)$ is indistinguishable from the case where the edge (u^*, v^*) has not yet been made.

Since U_i is equally distributed for all $i \in [N]$ and independent of b^*, it also holds

$$\begin{aligned}\Pr[V \wedge U_i \mid b^* = 0] &= \Pr[V \mid U_i \wedge (b^* = 0)] \Pr[U_i \mid b^* = 0] \\ &= \Pr[V \mid U_i \wedge (b^* = 0)] \Pr[U_i] \\ &= \Pr[V \mid U_{i-1} \wedge (b^* = 1)] \Pr[U_{i-1}] \\ &= \Pr[V \mid U_{i-1} \wedge (b^* = 1)] \Pr[U_{i-1} \mid b^* = 1] \\ &= \Pr[V \wedge U_{i-1} \mid b^* = 1].\end{aligned} \tag{5}$$

This implies

$$\begin{aligned}&\sum_{j=1}^{\delta} \sum_{i=1}^{N-1} \mathsf{Pr}[A' \to 0 \mid \varDelta_j \wedge V \wedge U_i \wedge (b^* = 0)] \cdot \mathsf{Pr}[\varDelta_j \mid V \wedge U_i \wedge (b^* = 0)] \cdot \mathsf{Pr}[V \wedge U_i \mid b^* = 0] \\ &= \sum_{j=1}^{\delta} \big(\mathsf{Pr}[A' \to 0 \mid \varDelta_j \wedge V \wedge U_1 \wedge (b^* = 0)] \cdot \mathsf{Pr}[\varDelta_j \mid V \wedge U_1 \wedge (b^* = 0)] \cdot \mathsf{Pr}[V \wedge U_1 \mid b^* = 0] \\ &\qquad + \sum_{i=2}^{j} \mathsf{Pr}[A' \to 0 \mid \varDelta_j \wedge V \wedge U_{i-1} \wedge (b^* = 1)] \cdot \mathsf{Pr}[\varDelta_j \mid V \wedge U_{i-1} \wedge (b^* = 1)] \cdot \mathsf{Pr}[V \wedge U_{i-1} \mid b^* = 1] \big)\end{aligned} \tag{6}$$

Thus, when computing the difference of the probabilities of $(A' \to 0) \wedge V \wedge U$ conditioned on $b^* = 0$ and $b^* = 1$, respectively, most terms cancel, except for the terms for the two extreme cases $U_1 \wedge (b^* = 0)$ and $U_j \wedge (b^* = 1)$.

$$\begin{aligned}\Pr[(A' \to 0) \wedge & U \wedge V \mid b^* = 0] - \Pr[(A' \to 0) \wedge U \wedge V \mid b^* = 1] \\ &= \sum_{j=1}^{\delta} (\mathsf{Pr}[(A' \to 0) \wedge \Delta_j \wedge V \wedge U_1 \mid b^* = 0] - \mathsf{Pr}[(A' \to 0) \wedge \Delta_j \wedge V \wedge U_j \mid b^* = 1]) \qquad (7) \\ &= \mathsf{Pr}[(A' \to 0) \wedge V \wedge U_1 \mid b^* = 0] - \mathsf{Pr}[(A' \to 0) \wedge V \wedge U_{\mathsf{last}} \mid b^* = 1],\end{aligned}$$

where U_{last} denotes the event that u^* is the source of the last encryption query incident on v which A makes. If V and U_1 happen and $b^* = 0$, then the game simulated to A is exactly the same as the real game GSD'_0; similarly, if V and U_{last} happen and $b^* = 1$, then the game simulated to A is exactly the same as the random game GSD'_1. In other words,

$$\Pr[A' \to 0 \mid V \wedge U_1 \wedge (b^* = 0)] = \Pr[A(\mathsf{GSD}'_0) \to 0]$$

and

$$\Pr[A' \to 0 \mid V \wedge U_{\mathsf{last}} \wedge (b^* = 1)] = \Pr[A(\mathsf{GSD}'_1) \to 0]. \qquad (8)$$

Now, as long as A does not issue the challenge query, the games simulated to A are identically distributed in both cases[19] $V \wedge U_1 \wedge (b^* = 0)$ and $V \wedge U_{\mathsf{last}} \wedge (b^* = 1)$: All edges incident on v^* encrypt the same seed (s in the first case and s' in the latter), and both seeds associated with v^* were neither queried to H nor reveiled by a corruption or challenge query. Hence, v^* looks just the same as any other potential challenge node and we have

$$\Pr[V \mid U_1 \wedge (b^* = 0)] = \Pr[V \mid U_{\mathsf{last}} \wedge (b^* = 1)] \geq \frac{1}{N}. \qquad (9)$$

Since we're guaranteed that the challenge node will not be a source node, we also have that

$$\Pr[U_1 \mid b^* = 0] = \Pr[U_{\mathsf{last}} \mid b^* = 1] \geq \frac{1}{N}. \qquad (10)$$

Combining equations (1)-(10), we finally arrive at

$$\begin{aligned}\Pr[(A' \to 0) \wedge & U \wedge V \mid b^* = 0] - \Pr[(A' \to 0) \wedge U \wedge V \mid b^* = 1] \\ &= \Pr[A' \to 0 \mid V \wedge U_1 \wedge (b^* = 0)] \cdot \Pr[V \wedge U_1 \mid b^* = 0] \\ &\quad - \Pr[A' \to 0 \mid V \wedge U_{\mathsf{last}} \wedge (b^* = 1)] \cdot \Pr[V \wedge U_{\mathsf{last}} \mid b^* = 1] \\ &= (\Pr[A' \to 0 \mid V \wedge U_1 \wedge (b^* = 0)] - \Pr[A' \to 0 \mid V \wedge U_{\mathsf{last}} \wedge (b^* = 1)]) \cdot \Pr[V \wedge U_1 \mid b^* = 0] \\ &\geq \frac{1}{N^2} \cdot \big(\mathsf{Pr}[A(\mathsf{GSD}'_0) \to 0] - \mathsf{Pr}[A(\mathsf{GSD}'_1) \to 0]\big).\end{aligned}$$

This proves the claim. □

Corollary 2. *Let the event E be defined as in the proof of Theorem 3, and A an arbitrary GSD adversary which triggers E with probability ϵ. Then there exists an IND-CPA adversary A' with advantage*

$$\Pr[A' \to 0 \mid b^* = 0] - \Pr[A' \to 0 \mid b^* = 1] \geq \frac{\epsilon}{N^2} - \frac{m}{N 2^{\ell}}.$$

Proof. We first apply Lemma 5 to construct an adversary against GSD" with advantage $\frac{\epsilon}{N} - \frac{m}{2^{\ell}}$. Then we apply a very similar proof as for Lemma 6 to construct an IND-CPA adversary: This proof is exactly the same with the only difference being that GSD" is less adaptive, i.e. A commits to the challenge node at the beginning of the game. This means that A' does not have to guess v^* and the event V is always true. Thus, we only lose an additional factor $1/N$ due to guessing u^*. Finally, we note that, since A' is sampling the keys itself for all nodes except u^*, the increase in the number of nodes from N to $N + 1$ due to Lemma 5 can be ignored, since A' does not actually need to create the additional node, since it knows the seed s_{v^*}. □

[19] Note that not all encryption queries incident on v might have been issued at this point, but this does not affect the analysis.