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

Karen Klein*♯§, Guillermo Pascual-Perez*♯♭, Michael Walter*♯§, Chethan Kamath, Margarita Capretto†, Miguel Cueto*, Ilia Markov*, Michelle Yeo*♭, Joël Alwen‡, Krzysztof Pietrzak*§
{kklein, gpascual, mwalter}@ist.ac.at

*IST Austria, †Universidad Nacional del Rosario, ‡Wickr Inc.

2021 IEEE Symposium on Security and Privacy (SP) | 978-1-7281-8934-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SP40001.2021.00035

***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). TreeKEM is the currently considered candidate by the IETF MLS working group, but dynamic group operations (i.e. adding and removing users) can cause efficiency issues. In this paper we formalize and analyze a variant of TreeKEM which we term Tainted TreeKEM (TTKEM for short). The basic idea underlying TTKEM was suggested by Millican (MLS mailing list, February 2018). This version is more efficient than TreeKEM for some natural distributions of group operations, we quantify this through simulations.**

Our second contribution is two security proofs for TTKEM which establish post compromise and forward secrecy even against adaptive attackers. The security loss (to the underlying PKE) in the Random Oracle Model is a polynomial factor, and a quasipolynomial one in the Standard Model. Our proofs can be adapted to TreeKEM as well. Before our work no security proof for any TreeKEM-like protocol establishing tight security against an adversary who can *adaptively* choose the sequence of operations was known. We also are the first to prove (or even formalize) *active* security where the server can arbitrarily deviate from the protocol specification. Proving fully active security – where also the users can arbitrarily deviate – remains open.

I. Introduction

Messaging systems allow for asynchronous communication, where parties need not be online at the same time. 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 end-to-end privacy and authenticity but, by having parties perform regular key updates, also 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 (usually all messages prior to the last key update) remain private. In turn, PCS guarantees that even if a party was compromised and its state leaks, normal protocol execution after the compromise ensures that eventually (usually 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*[1] scaling to larger groups is an important but challenging open problem. Designing such a protocol is the ongoing focus of the IETF working group Message Layer Security (MLS) [1].

Instead of constructing a messaging scheme directly, a modular approach 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 (CGKA), a generalisation for groups of their Continuous Key Agreement (CKA). Such a primitive can then be used to build a group messaging protocol as in [2].

A. 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. 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 a CGKA, we will assume that such an infrastructure exists.

Apart from initialising a group, CGKA allows any group member ID_i to *update* its key. Informally, after an Update[2] 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 party, 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

♯The first three authors contributed equally to this work.

§Funded by the European Research Council (ERC) under the European Union's Horizon2020 research and innovation programme (682815-TOCNeT).

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

[1]The meaning of efficient here will be determined by what can be implemented and receive adoption by the general public, but we think of it as having message sizes (poly)logarithmic in the size of the group.

[2]We use capital letters to refer to the operations (as opposed to verbs).

– 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.

The formalization of CGKA is fairly recent, having first been introduced in [3]. In particular, the MLS working group predates it, which only complicates any account of its development. We try to give a brief overview below. Up to the writing of this paper, the MLS protocol has seen 9 versions, through which we can find two different CGKA protocols: ART and TreeKEM, both of which were incorporated as candidates in the initial MLS protocol draft. ART was later removed in the second version of the protocol, with TreeKEM (which has seen several modifications throughout the different versions) being the current candidate. Accordingly, we will refer to the CGKA construction underlying version X of the MLS draft as TreeKEMvX. We will also loosely use the term TreeKEM when referring to aspects that are not unique to specific versions or when there is no ambiguity.

1) 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.* [4]. This protocol (as well as TreeKEM and the protocol formalized in this paper) identifies the group with a binary tree where edges are directed and point from the leaves to the root.[3] 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 size of the group.

The authors prove the ART protocol secure even against adaptive adversaries. However, in this 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.

[3]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.

2) TreeKEM: The TreeKEM proposal [5], [6] 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. We will refer to this as "encrypting v to u". Below we will outline a slightly simplified construction, close to TreeKEMv7, which will later ease the understanding of the protocol here proposed.

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 their public keys.) 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, she just samples a fresh path starting at a currently not occupied leaf, using ID_j's key as the new leaf node, and communicates the new keys to ID_j. This process can be optimized if the keys are derived hierarchically, from a hash chain of seeds, so that a single seed needs to be encrypted to each party.

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 not on this path, thus not replaced with the Update, the adversary will potentially still be able to compute the new group key, so the 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 declares any nodes violating the invariant as 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. The reason one can still hope for TreeKEM's efficiency to not degrade too much and stay close to logarithmic 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.

The protocol studied in this paper builds closely on the one just outlined. For completeness, we mention that the design of TreeKEMv9 differs in essentially two aspects. First, operations are not executed standalone, but bundled into groups: users can at any point *propose* an operation, not having any impact on the group state; then, a user ID_j can collect those proposals and execute them at once in a *Commit*, which includes an update of ID_j's path, and moves the group forward into a new epoch. This allows e.g. for ID_i to propose an Update by just sending their new leaf public key and waiting for someone else to commit that proposal (which will in turn blank ID_i's path). Second, Adds no longer involve blanking: a new user's leaf node will be directly connected to the root, and progressively pushed down the tree as users within the appropriate subtree commit. In particular, the initialization of the tree will now consist of a Commit including Add proposals for each of the group members. Since none of these aspects help in the understanding of the proposed protocol, we omit the details and refer the reader to the MLS draft [6].

B. *Our Contribution*

In this work we formalize an alternative CGKA design, stemming from TreeKEM, first proposed by Millican on the MLS mailing list on February 2018[4], which we call Tainted TreeKEM, or simply TTKEM. Further, we show it to be more efficient than TreeKEM on certain realistic scenarios and prove it to satisfy a comprehensive security statement 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.

1) 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. The rationale for blanking is to

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

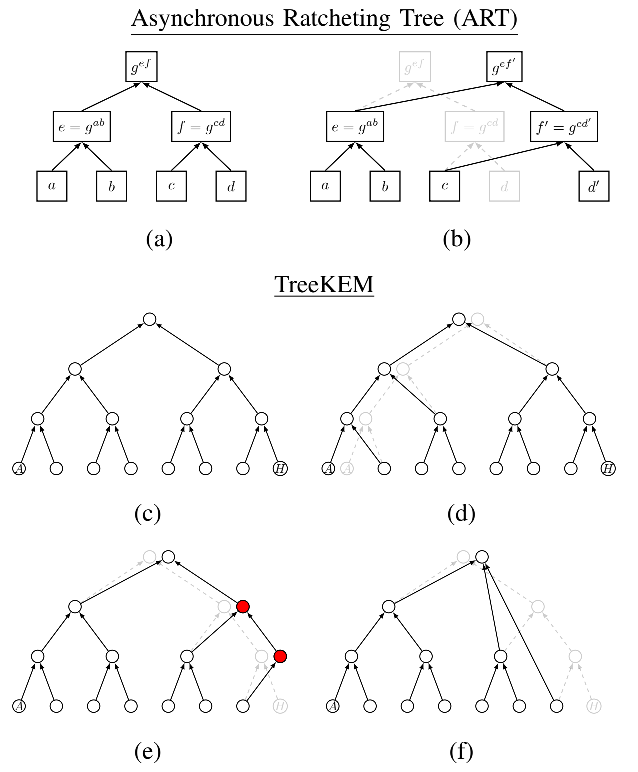

Fig. 1: **Top**: Illustration of an Update in the ART protocol. The state of the tree changes from (a) to (b) when Dave (node d) updates his internal state to d'. **Bottom**: Update and Remove in TreeKEM and TreeKEM with blanking. The state of a completely filled tree is shown in (c). The state changes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

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).

a) 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 it must get updated.

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

b) 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. In Section II-D 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 for this is that the alternative metric of measuring the number of ciphertexts a party needs to download to process an operation is not as relevant, since, all protocols considered, this number will be logarithmic in the worst case.[5]

2) 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. In particular, this constitutes the first adaptive security proof for any TreeKEM-related protocol. Moreover, both the security statement and the proof can be easily adapted to TreeKEM. We elaborate in the following section.

C. The Adversarial Model

We anticipate an adversary who works in rounds, in each round it may adaptively choose an action, including start/stop corrupting a party, instruct a party to initalize an operation or relay a message. 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 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.

We now discuss different possible restrictions on the adversary corresponding to qualitatively different levels of security.

a) Adaptiveness: The literature distinguishes between *selective* and *adaptive* adversaries. In the selective case, an adversary is required to make all or some of its choices (here this means the sequence of operations and which key it is going to break) at the beginning of the security experiment, without seeing any public keys or the results of previous actions. While it is often more convenient to prove security in this setting, it is clearly unrealistic, since in the real world adversaries may adjust their behaviour based on what they observe during the attack. So obviously, security against adaptive adversaries is desirable. There is a generic reduction from selective to adaptive adversaries that simply guesses what the adversary may choose (this is the approach effectively taken in [7]). However, this involves a loss in the advantage that is exponential (or even superexponential) in the size of the group. This means that in order to *provably* achieve meaningful security, one needs to set the underlying security parameter linear in the group size, which results in the Update messages having size linear in the group size (since they usually consist of encryptions of secret keys). But the trivial construction based on pairwise channels also has message size that is linear in the number of group members, so such a security proof defeats the whole purpose of the protocol: having small Update messages! The adversaries we consider are adaptive while the security loss we achieve is only quasipolynomial (or even polynomial) in the standard model (in the ROM, resp.; see details below).

b) Activeness: One can classify adversaries with respect to their power to interact with the protocol during the attack. For example, the weakest form of adversary would be a *passive* adversary, i.e. an eavesdropper that only observes the communication but does not alter any messages. While the strongest notion would be an *active* adversary who can behave completely arbitrarily. In this work we consider "partially" active adversaries who can arbitrarily schedule the messages of the delivery server, and thus force different users into inconsistent states. But we do not consider adversaries who can arbitrarily deviate and for example use secret keys of corrupted parties to create malformed messages. Restricting to partially active adversaries is fairly common in this setting [2], [8], [9] (also somewhat implied by the model of [10], where communication must halt after an active attack). Achieving or even defining meaningful security against fully active adversaries is the subject of ongoing research [11].

c) Forward Secrecy: FS (and PCS) are standard notions expected to hold in modern messaging protocols. However, in contrast to the two-party setting, formalizing FS in the group setting is more nuanced. One natural notion is to require that a key is secure if all parties have performed an update

[5]There is, however, 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.

before being corrupted. This is the notion considered in [7] and the one we adopt here and call it *standard FS*. In contrast, [3] defines a stronger notion we refer to as *strong FS*. It requires keys to be secure as soon as possible subject to not violating basic completeness of the CGKA protocol. However, this is only required in executions where protocol packets are delivered in the same order to all group members.[6] The construction in [3] in fact achieves strong FS, but only for adversaries that are much less active than ours. We provide some details in the next section.

1) The safe predicate: Providing PCS and FS requires to clearly define which keys we expect to be secure given a sequence of adversarial actions. Given the asynchronous setting where group members might be in different states, and an active adversary that may force users into inconsistent states, this is quite involved. Note that group members might even have different views of who is currently a member of the group. We give a compact and intuitive predicate that captures exactly what PCS and FS guarantees TTKEM provides.

2) The reduction in the standard model via piecewise guessing: Recall that there is a trivial reduction between selective and adaptive adversaries that simply guesses the necessary information and fails if the guess was incorrect. This loses an exponential factor in the amount of information that needs to be guessed. Jafargholi et al. [12] proposed a general framework (often refered to as *piecewise guessing*) that allows to reduce this loss under certain conditions. The resulting loss depends on the graph structure that naturally arises from the security experiment. Applying the framework in the obvious way (which already requires non-trivial effort) we achieve a quasipolynomial security loss $\approx (Q \cdot n)^{2\log(n)}$, 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) against partially active adversaries. Using a more careful analysis and taking the more restrictive structure of the queries and the graph constructed in the TTKEM security game into account, we can improve this to $\approx Q^{\log(n)}$. Our proof relies on [12] and requires familiarity with the framework, but is fully rigorous. We note that all steps of the proof strategy also apply to TreeKEM, and so an equivalent proof for it would easily follow.

3) 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 TTKEM.

This proof is very different from the proof in the standard model and does not use the piecewise guessing framework. Some of the techniques resemble a security proof of Logical Key Hierarchies (cf. Section I-D) by Panjwani [13], but otherwise the proof is entirely self-contained and novel. Again, our proof can also be applied to TreeKEM. As a sidenote, we prove and employ a new result on a *public-key* version of *generalized selective decryption* (GSD, an abstraction of security experiments involving encryptions of keys) in the ROM, which we believe to be of independent interest.

D. Related Work

The basic idea of TreeKEM can be traced back to Logical Key Hierarchies (LKH) [14], [15], [16]. 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 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 [13]. Unfortunately, there are several important differences between LKH that do not allow us to simply rely on [13] to prove TTKEM or TreeKEM secure: 1) their proof is in the symmetric key setting, while we are using public key encryption; 2) their proof 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 [13] does not take into account (even though it had already been proposed in optimized versions of LKH [16]) and it is a priori unclear how this affects the proof; 4) we are also interested in proving security in the ROM, which, as we show, gives tighter bounds.

Since the appearance of the double ratchet algorithm [17], implemented in applications like Signal or Whatsapp, secure messaging has received a lot of attention, particularly in the two party case [18], [10], [19], [9], [20], [2]. In the group setting, the main example of such a protocol is TreeKEM [5], [6], currently in development by the IETF MLS working group. Its predecesor was the ART protocol [4], 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.*[8], and Weidner [21] explored a variant of TreeKEM allowing for less reliance on the server for correctness. Finally, in a follow-up work, Alwen *et al.* [11] study the security of CGKA protocols against insider attacks.

rTreeKEM: Recently, Alwen *et al.* [3] introduced another variant of TreeKEM, termed re-randomized TreeKEM (rTreeKEM). Since their paper structure shares similarities with ours, we will discuss the differences between them.

First, it should be noted that the aims of the protocols are very different: while TTKEM seeks to improve the efficiency of TreeKEM by removing the need for blanks, rTreeKEM's focus is on improving its forward secrecy guarantees to achieve strong FS. However, we see no reason why one could not combine both protocols, endowing TTKEM with strong

[6]Going even further, the (efficient but impractical) CGKA protocols of [11] enjoys *optimal FS*. That is keys must become secure as soon as possible for *arbitrary* delivery order. In fact, 2 of their protocols enjoy optimal FS even against adversaries that can arbitrarily manipulate and generate traffic; a type of active security even stronger than the one considered in this work.

	Constraints on PKE	Forward Secrecy	Adversary	Tightness	
				Selective	Adaptive
ART [4]	ElGamal (or any other with a contributive NIKE)	Standard	Passive	$O(n)$ (ROM)	$\Omega\left((nQ)^n\right)$ (ROM)
TreeKEM [6]**	None	Standard	Passive	$\mathcal{O}(n)$ (SM)	$\Omega\left((nQ)^n\right)$ (SM) *
rTreeKEM [3]	UPKE	Strong	Passive	$\mathcal{O}(n)$ (ROM)	$\Omega\left((nQ)^n\right)$ (ROM) *
TTKEM (this work)	None	Standard	Partially active	$\mathcal{O}(n)$ (SM,ROM)	$\mathcal{O}\left(n^2 Q^{\log(n)}\right)$ (SM) $\mathcal{O}\left((nQ)^2\right)$ (ROM)

TABLE I: Table depicting the different security levels satisfied by CGKA protocols. The first two columns correspond to protocol characteristics, and the right-most three to the best known proofs. SM and ROM stand Standard and Random Oracle Models, respectively. (*) These would follow from the selective proof via a straightforward complexity leveraging argument. Such an argument is implicit in the proof of [4]. (**) [3] provides a sketch for a proof against passive adaptive adversaries with a quasi-polynomial loss in the SM; this paper suggests proofs against part. active, adaptive adversaries both in the SM and ROM, with the same tightness as the ones for TTKEM.

FS. Moreover, it seems plausible that the proof techniques developed in this work can also be applied to the rTreeKEM construction or to the combination of the two.

Second, their work already defines CGKA as an abstraction of the main problem TreeKEM aims to solve. We use their completeness notion, but add a *Confirm and Deliver* algorithm to their definition. The reason for this is that we work in the more general model that allows a malicious delivery server, i.e. the adversary can reorder and withhold messages at will. The model in [3] requires the delivery server to be basically honest: the server can delay, but never send inconsistent messages to parties, i.e. the adversary in [3] is almost passive.

Last, both works provide security proofs, albeit these differ considerably. Their paper provides proofs for both TreeKEM and rTreeKEM with a polynomial security loss, although these concern selective security only. They also sketch a security proof against adaptive adversaries losing a quasi-polynomial factor (for TreeKEM in the standard model, for rTreeKEM in the ROM). In contrast, we give formal proofs for the *adaptive* security of TTKEM with only polynomial loss in the ROM and quasi-polynomial in the standard model; and, as mentioned, against a stronger partially active adversary. Also, proofs with the same bounds would follow for TreeKEM.

E. Impact on MLS

As of writing, the current version of the MLS draft (MLS v9) differs substantially from TTKEM, mainly due to the Proposal-Commit structure. However, it should be noted that TTKEM can be cast in that same fashion, as it is indeed done in [11]. As with TreeKEM, the application of this framework would bring an efficiency tradeoff that should be studied carefully and which we leave for further work, though noting the challenge in doing so without real world data. As for our security proofs, a security proof for TreeKEM follows from the one given in the paper, so we believe this work to be of relevance to the MLS community.

II. Description of TTKEM

A. 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* $\mathsf{G} = (\mathsf{ID}_1, \ldots, \mathsf{ID}_n)$*. Party* ID_1 *creates a new group with membership* G *by running:*

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

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 (incl.*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* $(\gamma, I) \leftarrow \mathsf{key}(\gamma)$*.*

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.

B. 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 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 Δ; a secret/public key pair derived deterministically as $(pk, sk) \leftarrow \mathsf{Gen}(\Delta)$; a credential (leaf nodes only); and a tainter ID (all nodes but leaves and root). The root has no associated public/secret key pair, instead its seed is the current group key.

To achieve FS and PCS, and to manage group membership, it is necessary to constantly renew the secret keys used in the protocol. We will do this through the group operations **Update**, **Remove** and **Add**. We will use the term *refresh* to refer to the renewal of a particular (set of) key(s) (as opposed to the group operation). Each group operation will refresh a part of the tree, always including the root and thus resulting in a new group key which can be decrypted by all members of the current group. Users will also have a list of Initialization Keys (init keys) stored in some key-server, widely available and regularly updated, and 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. We assume that a party will only process operations issued by parties that (at the time of issuing) shared the same view of the tree. This can easily be enforced by adding a (collision-resistant) hash of the operations processed so far [10], [9][7]. Furthermore, group members will have a partial view of 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 updated group state resulting from the last issued group operation while they wait for confirmation.

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

[7]For efficiency reasons one could use a Merkle-Damgård 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.

a) Tainting.: Whenever party ID_i refreshes a node not lying on their path to the root, that node becomes *tainted* by ID_i. Whenever a node is tainted by a party ID_i, that party has potentially had knowledge of its current secret in the past. So, if ID_i was corrupted in the past, the secrecy of that value is considered 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 tainted ID of a node is determined by the following simple rule: after ID issues an operation, all refreshed nodes on ID's path become untainted; in turn, all refreshed nodes *not* on ID's path become tainted by ID.

b) Hierarchical derivation of updates.: When refreshing a whole path we sample a seed Δ_0 and derive all the secrets for that path from it. This way, we reduce the number of decryptions needed to process the update, as parties only need to recover the seed for the "lowest" node that concerns them, and then can derive the rest locally. To derive the different new secrets we follow the specification of TreeKEMv9 [6]. 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 as$\Delta_{i+1} := H_1(\Delta_i)$ and $(sk_i, pk_i) \leftarrow \mathsf{Gen}(H_2(\Delta_i))$ 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. Hence, we partition the set of all the nodes to be refreshed into paths and use a different seed for each path. Any unambiguous ordered optimal partition will suffice. 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). An example is provided in the appendix.

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

C. 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 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 operation type, encryptions of the new seeds, any

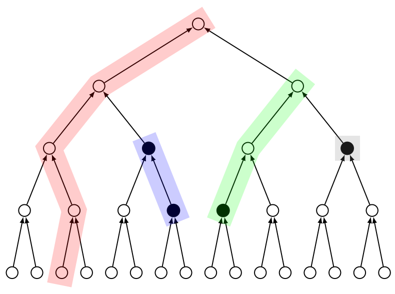

Fig. 2: Path partition resulting from an update by Charlie (3rd leaf node), with nodes tainted by him shown in black. To process it the grey node must be updated before the green path and the blue path before Charlie's (in red).

new public keys, and a hash of the transcript so far, ensuring consistency. A more detailed description, as well as pseudo-code for the distinct operations is presented in B1.

a) 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 $\mathcal{T}$.

b) 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[8], 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 $\mathcal{T}$.

c) 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 copaths and sends this information to the server.

d) Remove.: To remove ID_j, user ID_i performs an Update as if it was ID_j, refreshing all 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 could request for someone to remove her and delete her state.

e) 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, he updates the current state γ to γ' or simply deletes γ', respectively

[8]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.

D. Comparison with Blanking

In terms of security there is 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[9] 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.

[9]We compare TTKEM with the most recent version TreeKEMv9.

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[10]. 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 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 [22]. 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 [23]. 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.

[10][MLS] Cost of the partial-tree approach. Richard Barnes {rlb@ipv.sx} 01 October 2018 https://mailarchive.ietf.org/arch/msg/mls/hhl0q-OgnGUJS1djdmH1JBMqOSY/

III. Security

We will prove security for TTKEM against fully adaptive, partially active adversaries, even when group members are in inconsistent states. In section III-A we present the security game we consider and in section III-B 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.

A. 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* $(\mathsf{ID}_1, \dots, \mathsf{ID}_\ell)$*. 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.*

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*

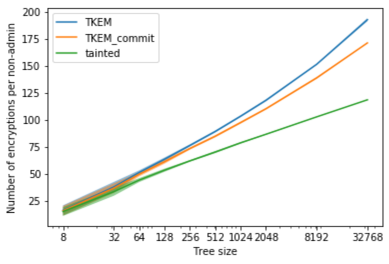

Fig. 3: Cost for non-administrators

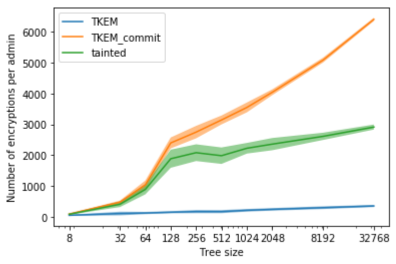

Fig. 4: Cost for administrators

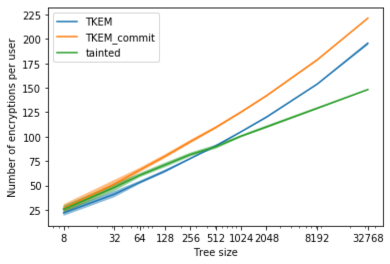

Fig. 5: Average cost per user

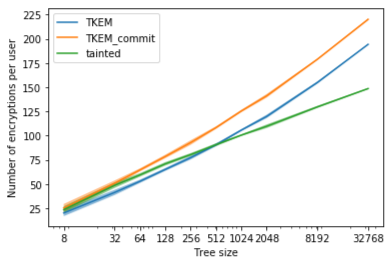

Fig. 6: Updaters follow uniform dist.

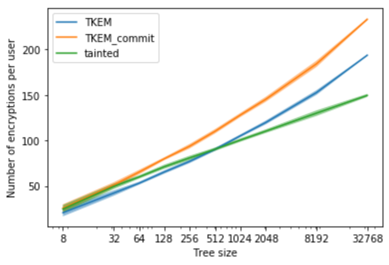

Fig. 7: Updaters follow Zipf dist.

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*

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

B. 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[11] at time q^* must not be compromised as long as these init keys are part of their state. However, defining a reasonable safe predicate in terms of allowed sequences of actions 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. 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

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

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.

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

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^*.

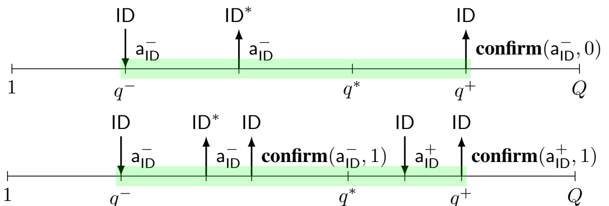

Fig. 8: A schematic diagram showing the critical window for a user ID in the view of another user ID^* with respect to query q^*. An arrow from a user to the timeline is interpreted as a request by the user, whereas an arrow in the opposite direction is interpreted as the user processing the message. 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*

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

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* $\mathsf{a}^* = \textbf{update}(\mathsf{ID}^*)$ *then we require* ID^* *to be safe w.r.t. the window* $[q^*, q^+]$.

C. 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 1) — this fact will be crucial in the next section (§III-E) where we argue the security of TTKEM using the framework of Jafargholi et al. [12]. 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.

a) 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 §II). 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.

b) 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 the key lemma which connects the safe predicate to the challenge graph of TTKEM; a proof can be found in the full version of this paper.

Lemma 1. *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.*

D. 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. [12], with which we will assume familiarity throughout this section and, particularly, in Theorem 4. 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 [12], we will define these hybrid games via the so-called *reversible black pebbling* game, introduced by Bennett [24], 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 (L 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 $\mathfrak{P} = (\mathcal{P}_0, \dots, \mathcal{P}_L)$ 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:*

- *$\mathcal{P}_\ell = \mathcal{P}_{\ell-1} \cup \{v\}$ or $\mathcal{P}_\ell = \mathcal{P}_{\ell-1} \setminus \{v\}$,*
- *for all $u \in \mathrm{parents}(v)$: $u \in \mathcal{P}_{\ell-1}$.*

By Lemma 1, 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*

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

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 2. *Let $\mathfrak{P} = (\mathcal{P}_0, \dots, \mathcal{P}_L)$ 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*[12].

A proof of this lemma can be found in the full version of this paper. 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 security loss. However, the framework of Jafargholi et al. [12] allows to do significantly better:

Theorem 3 (Framework for proving adaptive security, informal [12])**.** *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 $\mathsf{H}_{\mathsf{real}} := \mathsf{H}_0, \mathsf{H}_1, \dots, \mathsf{H}_L := \mathsf{H}_{\mathsf{random}}$ 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 [25, Algorithm 1], which uses $L = n^2$ steps and only $2\log(n) + 1$ pebbles[13], implies a security reduction for TTKEM with only

[12]Technically, the t in Lemma 2 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.

[13]Although the original Lemma 3 in [25] 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 trees leads to a bound of $2\log n + 1$.

a quasipolynomial loss in security.

Theorem 4. *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 $\mathfrak{P} = (\mathcal{P}_0, \dots, \mathcal{P}_L)$ be the recursive pebbling strategy for binary trees from [25], 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 $M = (\log(n) + 2) \cdot \log Q$ bits. The claim then follows by Lemma 2 and Theorem 3.

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. □

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.

E. 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 [13])**.** *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*

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

We will apply the following general result for our version of GSD, which could be of independent interest; a proof can be found in the full version of this paper.

Theorem 5. *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 (ϵ, t)-IND-CPA secure and H is modelled as a random oracle, then Π' is $(\tilde{\epsilon}, t)$-adaptive GSD secure, where $\tilde{\epsilon} = \epsilon \cdot 2N^2 + (mN)/(2^{l-1})$, with N the number of nodes, m the number of oracle queries to H and l the seed length.

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; a proof can be found in the full version of this paper.

Theorem 6. *If the encryption scheme 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 $\epsilon = \tilde{\epsilon} \cdot 8(nQ)^2 + \mathsf{negl}$.*

We remark that, similarly to the previous proof, one can easily adapt it to the case of TreeKEM (with blanking).

IV. Conclusion

We formalized and analyzed a proposed modification to TreeKEM, the Continuous Group Key Agreement (CGKA) algorithm that as of September 2020 is being considered for standardization by the IETF "Message Layer Security" working group. First, we showed that the modification, termed TTKEM, has the potential to achieve better efficiency than TreeKEM for large groups in natural settings; and is therefore worth of further work and consideration. Second, we formulated a novel and intuitive security model against active and adaptive outsiders for a CGKA with Forward Secrecy and Post-Compromise Security. Third, we provided security proofs for TTKEM in both the standard and RO model, bounding the security loss (to the underlying PKE) by a quasipolynomial factor $Q^{\log(n)}$ and a polynomial factor $(Qn)^2$ respectively, where n is the group size and Q the total number of (update/remove/invite) operations. Our proof techniques can easily be extended to TreeKEM.

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 *EUROCRYPT 2019, Part I*, ser. LNCS, Y. Ishai and V. Rijmen, Eds., vol. 11476. Springer, Heidelberg, May 2019, pp. 129–158.

[3] J. Alwen, S. Coretti, Y. Dodis, and Y. Tselekounis, "Security analysis and improvements for the ietf mls standard for group messaging," in *Advances in Cryptology – CRYPTO 2020*, D. Micciancio and T. Ristenpart, Eds. Cham: Springer International Publishing, 2020, pp. 248–277.

[4] 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. [Online]. Available: https://doi.org/10.1145/3243734.3243747

[5] K. Bhargavan, R. Barnes, and E. Rescorla, "TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups," May 2018.

[6] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert, "The Messaging Layer Security (MLS) Protocol," Internet Engineering Task Force, Internet-Draft draft-ietf-mls-protocol-09, Mar. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-mls-protocol-09

[7] K. Cohn-Gordon, C. Cremers, L. Garratt, J. Millican, and K. Milner, "On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees," in *ACM CCS 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM Press, Oct. 2018, pp. 1802–1819.

[8] 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.

[9] D. Jost, U. Maurer, and M. Mularczyk, "Efficient ratcheting: Almost-optimal guarantees for secure messaging," in *EUROCRYPT 2019, Part I*, ser. LNCS, Y. Ishai and V. Rijmen, Eds., vol. 11476. Springer, Heidelberg, May 2019, pp. 159–188.

[10] F. B. Durak and S. Vaudenay, "Bidirectional asynchronous ratcheted key agreement with linear complexity," in *IWSEC 19*, ser. LNCS, N. Attrapadung and T. Yagi, Eds., vol. 11689. Springer, Heidelberg, Aug. 2019, pp. 343–362.

[11] J. Alwen, S. Coretti, D. Jost, and M. Mularczyk, "Continuous group key agreement with active security," in *TCC 2020, Theory of Cryptography Conference, Durham, NC, USA, November*, 2020. [Online]. Available: https://eprint.iacr.org/2020/752

[12] Z. Jafargholi, C. Kamath, K. Klein, I. Komargodski, K. Pietrzak, and D. Wichs, "Be adaptive, avoid overcommitting," in *CRYPTO 2017, Part I*, ser. LNCS, J. Katz and H. Shacham, Eds., vol. 10401. Springer, Heidelberg, Aug. 2017, pp. 133–163.

[13] S. Panjwani, "Tackling adaptive corruptions in multicast encryption protocols," in *TCC 2007*, ser. LNCS, S. P. Vadhan, Ed., vol. 4392. Springer, Heidelberg, Feb. 2007, pp. 21–40.

[14] D. M. Wallner, E. J. Harder, and R. C. Agee, "Key management for multicast: Issues and architectures," Internet Draft, Sep. 1998, http://www.ietf.org/ID.html.

[15] C. K. Wong, M. G. Gouda, and S. S. Lam, "Secure group communications using key graphs," in *Proceedings of ACM SIGCOMM*, Vancouver, BC, Canada, Aug. 31 – Sep. 4, 1998, pp. 68–79.

[16] 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*, New York, NY, USA, Mar. 21–25, 1999, pp. 708–716.

[17] T. Perrin and M. Marlinspike, "The Double Ratchet Algorithm," https://signal.org/docs/specifications/doubleratchet/, 2016.

[18] M. Bellare, A. C. Singh, J. Jaeger, M. Nyayapati, and I. Stepanovs, "Ratcheted encryption and key exchange: The security of messaging," in *CRYPTO 2017, Part III*, ser. LNCS, J. Katz and H. Shacham, Eds., vol. 10403. Springer, Heidelberg, Aug. 2017, pp. 619–650.

[19] J. Jaeger and I. Stepanovs, "Optimal channel security against fine-grained state compromise: The safety of messaging," in *CRYPTO 2018, Part I*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10991. Springer, Heidelberg, Aug. 2018, pp. 33–62.

[20] B. Poettering and P. Rösler, "Towards bidirectional ratcheted key exchange," in *CRYPTO 2018, Part I*, ser. LNCS, H. Shacham and A. Boldyreva, Eds., vol. 10991. Springer, Heidelberg, Aug. 2018, pp. 3–32.

[21] Matthew A. Weidner, "Group Messaging for Secure Asynchronous Collaboration," Master's thesis, University of Cambridge, June 2019.

[22] 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*, vol. 106, no. 31, pp. 12 640–12 645, 2009. [Online]. Available: https://www.pnas.org/content/106/31/12640

[23] X. Gabaix, "Zipf's law for cities: An explanation," *The Quarterly Journal of Economics*, vol. 114, no. 3, pp. 739–7675, 1999. [Online]. Available: https://doi.org/10.1162/003355399556133

[24] C. H. Bennett, "Time/space trade-offs for reversible computation," *SIAM J. Comput.*, vol. 18, no. 4, pp. 766–776, 1989.

[25] G. Fuchsbauer, C. Kamath, K. Klein, and K. Pietrzak, "Adaptively secure proxy re-encryption," in *PKC 2019, Part II*, ser. LNCS, D. Lin and K. Sako, Eds., vol. 11443. Springer, Heidelberg, Apr. 2019, pp. 317–346.

Appendix

A. 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`.

B. 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 $i < j$.

1) 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.

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** $(\gamma, \mathsf{ID}, T)$
    $P_0 \leftarrow \gamma$.path($\mathsf{ID}$)
    $\{P_1, \ldots, P_n\} \leftarrow \gamma$.tainted-by($\mathsf{ID}$) *#refresh all paths from tainted nodes to root*
    **for** $i = n, \ldots, 0$ **do**
        $v_{i,0}, \ldots, v_{i,m} \leftarrow P_i$
        $\{\Delta_{i,0}, \ldots \Delta_{i,m}\} \leftarrow$ expand(gen-seed(), $m + 1$)
        **for** $p \in parents(v_{i,0})$ **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** $j = 1, \ldots, m$ **do**
            $T_{new\_seeds}.insert(\mathsf{Enc}_{\gamma.\mathtt{get\_pk}(\gamma.\mathtt{partner}(v_{i,j-1}))}(\Delta_{i,j}))$
            *refresh-node*$(\gamma, v_{i,j}, \Delta_{i,j}, T)$
```

```
**refresh-node** $(\gamma, v, \Delta, T)$
    **if** $v = v_{root}$ **then**
        $\gamma$.set_sk($v_{root}, \Delta$)
    **else**
        $(sk, pk) \leftarrow \mathsf{Gen}(H_2(\Delta))$
        $\gamma$.set_pk($v, pk$);
        $\gamma$.set_tainter($v$, me)
        $T_{new\_pks}.insert(pk)$
        **if** $v \in \gamma.path(\mathsf{ID})$ **then**
            $\gamma$.set_sk($v, sk$)
```

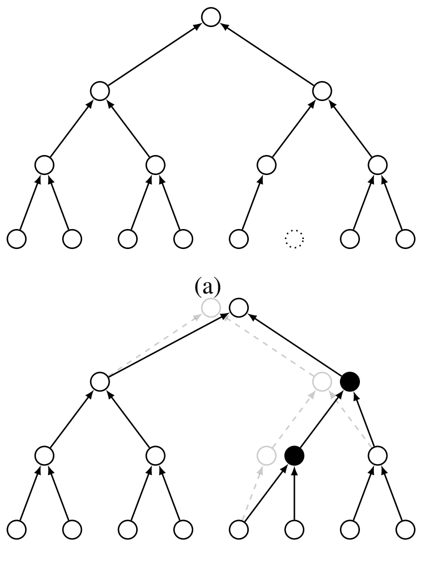

Fig. 9: Sample Add operation: (a) illustrates the state of the tree before Alice adds Frank (6th node) after which it turns into (b).

```
**add** $(\gamma, \mathsf{ID}, pk)$
    $\gamma' \leftarrow \gamma$
    $\gamma'.\texttt{add\_party}(\mathsf{ID}, pk)$
    $\{v_0, \ldots, v_d\} \leftarrow \gamma'.\texttt{path}(\mathsf{ID})$
    $sk \leftarrow \gamma'.\texttt{get\_sk}(\gamma'.\texttt{index}(\texttt{me}))$
    $r \leftarrow \$; \Delta \leftarrow h(sk, r)$
    $\{\Delta_0, \ldots, \Delta_d\} \leftarrow \texttt{expand}(\Delta, d+1)$
    *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**
            $T_{new\_seeds}.insert(\mathsf{Enc}_{\gamma.\texttt{get\_pk}(u)}\Delta_i)$
        *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**$(\gamma', W, T)$
```

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

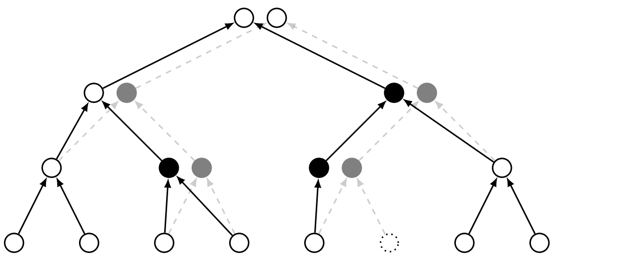
Fig. 11: Alice removes Frank (dotted) and in the process has to update his tainted nodes. Old state is again showed in gray.

H

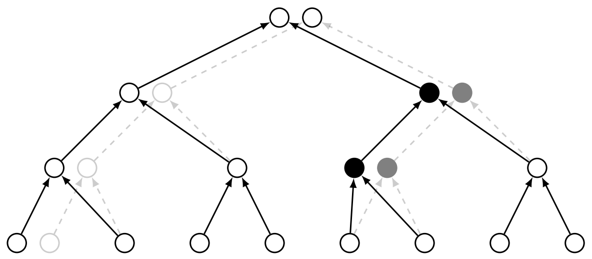
Fig. 10: A sample Update operation: Alice added Eve to the group which resulted in the tainted nodes (filled). Alice decided to later update herself. The state of the tree before the Update is in a lighter shade.

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

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** $p_l \neq \perp \wedge p_l \in \gamma.path(me)$ **then**
            $sk \leftarrow \gamma.\texttt{get\_sk}(p_l)$
        **else**
            $sk \leftarrow \gamma.\texttt{get\_sk}(p_r)$
        *update-path*$(\gamma, \{v_0, \ldots, v_n\}, \mathsf{Dec}_{sk}(enc), sender)$
```

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