Tighter Security for Group Key Agreement in the Random Oracle Model

Andreas Ellison and Karen Klein

ETH Zurich, Switzerland
{andreas.ellison,karen.klein}@inf.ethz.ch

Abstract. The Messaging Layer Security (MLS) protocol, recently standardized in RFC 9420 [6], aims to provide efficient asynchronous group key establishment with strong security guarantees. The main component of MLS, which is the source of its key efficiency and security properties, is a protocol called TreeKEM [9]. Given that a major vision for the MLS protocol is for it to become the new standard for messaging applications like WhatsApp, Facebook Messenger, Signal, etc., it has the potential to be used by a huge number of users. Thus, it is important to better understand the security of MLS and hence also of TreeKEM. In [1], TreeKEM was proven adaptively secure in the Random Oracle Model (ROM) with a polynomial loss in security by proving a result about the security of an arbitrary IND-CPA secure public-key encryption scheme in a public-key version of the Generalized Selective Decryption (GSD) security game [14]. In this work, we prove a tighter bound for the security of TreeKEM. We follow the approach in [1] and first introduce a modified version of the public-key GSD game better suited for analyzing TreeKEM. We then provide a simple and detailed proof of security for a specific encryption scheme, the DHIES scheme (currently the only standardized scheme in MLS), in this game in the ROM and achieve a tighter bound compared to the result in [1]. We then state the result on the security of TreeKEM implied by this bound, and give an interpretation of the result with protocol parameters used in practice.

Keywords: Messaging Layer Security · TreeKEM · Secure Messaging · Group Key-Agreement · Adaptive Security · DHIES

1 Introduction

We all rely on messaging applications like WhatsApp, Facebook Messenger, Signal, etc. in our daily lives and take it for granted that our messages will be transmitted securely, for some definition of “secure”. A common security feature expected from the protocol employed in a messaging application and known also to the general public is end-to-end encryption, i.e. that only the end users of a messaging session can read the messages being sent and the service provider or any party with access to the communication channel learns nothing of their contents. Another straightforward feature is that the protocol should work in an asynchronous setting: we would like to send messages even when the recipient is offline, and we expect them to receive the messsage once they come online. For this we must rely on a delivery service to store and deliver the messages. Of course also this delivery service should learn nothing about the contents of the messages.

There are two more advanced security features expected from messaging protocols today, both related to security in case a user is compromised:

- forward secrecy (FS): the compromise should not reveal the contents of old messages
- post-compromise security (PCS): after the user recovers from the compromise, new messages are secure once again

As a user may well not know that they have been compromised, ensuring PCS requires regularly updating the key material used for encryption (in a way that the information leaked in a compromise *before* the update does not suffice to compute encryption keys used *after* the update). The more often the key material is updated, the stronger the level of PCS that is achieved. Thus, updating the key material should be an efficient operation.

For messaging between two users, the Double Ratchet protocol [15], the main component of the so-called Signal Protocol, is a widely adopted solution used by major messaging applications such as Signal, WhatsApp, Facebook Messenger and more. It is well studied and achieves all of the above security guarantees [10]. For messaging in a group of more than two users, a straightforward solution is to maintain 1:1 communication channels using the Double Ratchet protocol between every pair of users and send messages to the group by sending them to every member individually. This achieves very strong security guarantees, but requires a number of encryption operations linear in the group size to send a message.

Another common solution is to use sender keys [5]: every user creates a symmetric key, their *sender key*, and distributes this sender key to every other user using 1:1 channels as before. A user sending a message then derives a symmetric encryption key for the message from their sender key, while continually updating their sender key (with each sent message) to provide FS. However, achieving PCS is costly: if a user is compromised, the sender keys of all users are leaked and recovering from the compromise requires each user to send a new sender key to every other user over the respective 1:1 channels, resulting in a number of operations linear in the group size per user and a quadratic number of operations in total. Moreover, dynamic group membership introduces additional complexity:

- adding a new member involves the new member sharing their sender key with all other group members
- removing a member requires distributing new sender keys in the group, just like recovering from a compromise

The Messaging Layer Security (MLS) protocol, recently standardized in [6], proposes a solution for group messaging with better efficiency and the same strong security guarantees as for the two-party case. Updating key material and adding or removing members can be achieved with a logarithmic number of operations (although the complexity may still degrade to linear in certain scenarios). At the core of MLS is a fairly recent primitive called a *continuous group key agreement* (CGKA) scheme [2] (this primitive was introduced only *after* the first draft of the MLS protocol). In essence, a CGKA scheme enables a group of users to agree on a *group key*, which they can then use to derive symmetric message encryption keys. This key must be indistinguishable from a random key for anyone outside the group eavesdropping on all communication. However, a CGKA scheme must also achieve FS and PCS, and support dynamic group membership. Hence, it must provide mechanisms for members to update their key material, add new users to the group and remove members from the group. Moreover, the scheme must work in the asynchronous setting with an untrusted service to deliver protocol messages.

The CGKA scheme used in the MLS protocol is called TreeKEM (initially proposed in [9]) and the majority of the literature on MLS is dedicated to analyzing TreeKEM or proposing better CGKA schemes as in [1,2,4,3]. The TreeKEM protocol has undergone multiple changes since its inception. In this work we refer to the version documented in RFC 9420. TreeKEM, as adopted from its predecessors, maintains a binary tree where every node in the tree has some associated secrets, every member of the group is associated with a leaf and the group key is derived from the root of the tree. Every member can compute the group key from their view of the tree. The group key can be updated and members added/removed with a number of operations logarithmic in the group size.

Given that the vision for the MLS protocol is for it to become the new standard for messaging protocols and that it has support from several large companies [11,12], it has the potential to be used by a huge number of users. Thus, understanding the security of MLS and hence also of TreeKEM is of great importance. This means having formal security guarantees about the security provided by TreeKEM (based on appropriate hardness assumptions). The first important step in this direction was the conception of the CGKA primitive and the accompanying definitions of security introduced in different works (for example [2,1]). Such definitions clarify what kind of adversaries we can provide security against and thus what kind of security one should expect from the scheme when using it in practice. Moreover, proofs of (reasonably tight) security under these definitions show what level of security we should expect from the scheme and serve as a guide to implementors on what values to choose for the security parameter. Proofs also provide strong justification that there are no flaws in the overall design of the scheme.

One choice that can be made when defining the security of a CGKA scheme is whether the adversary is modeled as *selective* or *adaptive*. In the former case, the adversary must provide all the interactions it will have with the protocol and when it will attempt to break the scheme at the beginning of the security game, while in the latter case the adversary can make its decisions based on responses from previous interactions. Clearly, the adaptive setting is much closer to how an attack would unfold in practice, so it is desirable to prove security against adaptive adversaries. However, achieving this without too much of a blow-up in the security loss is a challenge since one often resorts to guessing actions performed by the adversary.

The Generalized Selective Decryption (GSD) security game [14] was introduced precisely to analyze adaptive security for protocols based on a graph-like structure (as is the case with TreeKEM). It was initially defined for the private-key setting and later adapted to the public-key setting in [1]. The work in [1] proved a polynomial bound for the adaptive security of the public-key GSD game in the so-called Random Oracle Model (ROM) for an arbitrary IND-CPA secure public-key encryption scheme. This result implies a polynomial bound for the adaptive security of TreeKEM as a CGKA scheme as outlined in [1, Theorem 4] and subsequently proved in more detail in [3, Theorem 12].

1.1 Contributions

In this work, we formally prove the adaptive security of a specific public-key encryption scheme, the DHIES scheme, in a modified version of the public-key GSD game, adapted to better model TreeKEM, in the ROM. Focusing on the DHIES scheme allows us to achieve a tighter bound than the one in [1]. We then state the result on the security of TreeKEM implied by this bound, and give an interpretation of the result with protocol parameters used in practice. Moreover, we provide precise definitions for the syntax and security of propose and commit CGKA schemes and a high-level description of how the TreeKEM protocol can be instantiated with our definitions in the appendix.

1.2 Technical overview

The GSD game In the GSD security game, given an encryption scheme a graph, the *GSD graph*, is constructed by the challenger where every node in the graph is associated with a symmetric key in the private-key setting, or a public/private key pair in the public-key setting. The adversary can then request encryptions of a node's (secret) key under the (public) key of another node. In the public-key setting, such an *encryption query* also reveals the node's public key. This creates an *encryption edge* in the graph, directed from the node whose (public) key was used for encryption to the node whose key was encrypted. The adversary can also corrupt any node, which reveals its (secret) key and allows the adversary to compute the (secret) key of any other node reachable from the corrupted node in the graph by performing decryptions along the path to the other node. At the end of the game the adversary chooses a node to be challenged on, the *challenge node*. A coin is then tossed and the adversary is given either the (secret) key of the challenge node or a uniformly random (secret) key and it must guess which scenario it is in. The possible choices for the challenge node must of course be restricted to nodes whose keys were not compromised through a corruption, meaning that the challenge node should never be reachable from a corrupted node in the graph. Further restrictions are also necessary which we do not go into here. Figure 1 illustrates what an example GSD graph may look like.

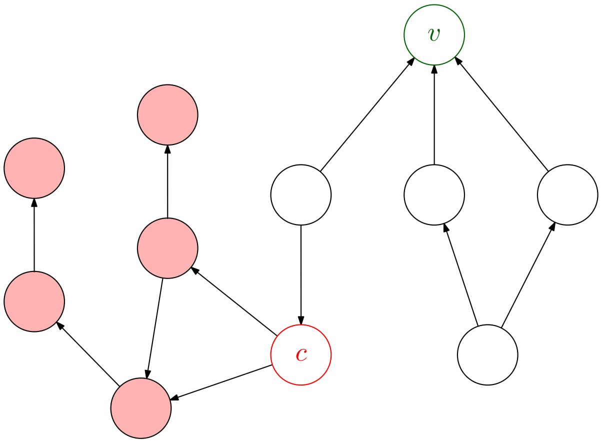


Fig. 1: An illustration of the GSD graph for an instance of the GSD game. The challenge node is v. The node c was corrupted, resulting in all nodes reachable from it being compromised, as marked with red color.

The TreeKEM protocol

Propose and commit syntax As a CGKA scheme, TreeKEM must support operations for updating the key material of a group member, adding a new user and removing a member. The syntax for these operations has changed over time. In the current version of MLS, the protocol uses so-called *proposals* and *commits*. Whenever a user would like have their key material updated (by someone else), add a new user or remove a group member, they create a corresponding *update*, *add* or *remove proposal*, respectively, and share this proposal with the group. Any group member can then create a *commit* to apply a set of proposals, create a new group key and update their key material in the process. The commit object includes (encrypted) information such that every group member can update their view of the group and compute the new group key.

TreeKEM dynamics As already outlined briefly, TreeKEM uses a full binary tree to model the group. Every user, associated with a leaf in the tree, maintains a synchronized view of the tree, though different users will know more about different parts of the tree. The group key is derived from the root of the tree. Every node n in the tree has an associated key pair (pk_n, sk_n) output by Π.Gen where Π is a public-key encryption scheme. All public keys are known to all users.

Let the *direct path* of a leaf be the path from the leaf's first parent to the root. Every user at a leaf knows the secret key of their leaf and, in the usual case, the secret keys of all nodes on their direct path, though we will see exceptions to this rule later. To illustrate the scheme and how commit operations are performed, we will consider of a group with users $A, B, \ldots, G$ and H, as depicted in Figure 2. In the following, we will use these labels for the users both to refer to the users themselves and to their nodes in the tree.

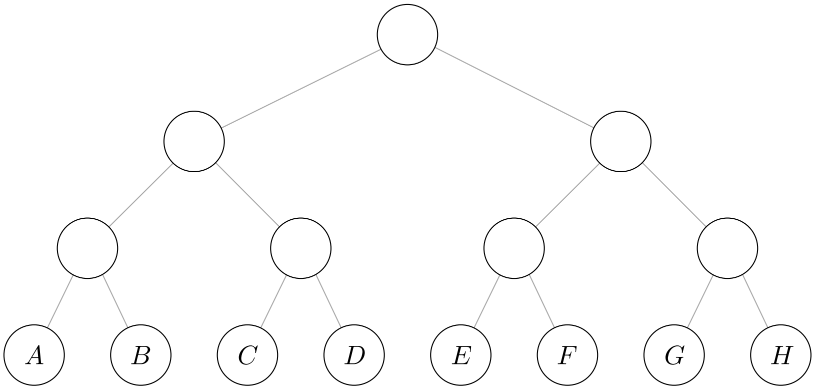


Fig. 2: Illustration of a group with users 8 users in the TreeKEM protocol.

Simple commits The idea behind this tree structure is that it allows for a user creating a commit with a new group key to share the new group key with the group using only a few encryptions, while still updating all the secrets the user knew in the tree in order to recover from a possible compromise (recall that in a PC-CGKA scheme a commit also updates the committer's key material). To illustrate how a commit is performed and how the new group key is computed, say user A performs a commit. First we only consider commits without any proposals. TreeKEM specifies two hash functions $H_{\text{gen}}, H_{\text{dep}} \colon \{0,1\}^{\rho(\eta)} \to \{0,1\}^{\rho(\eta)}$ where $\rho(\eta)$ gives the number of bits of randomness used by $\mathit{\Pi}.\text{Gen}(1^\eta)$. Let $d = 3$ the depth of user A. A will replace all the $d+1$ nodes on their path to the root (including their leaf) with new nodes $A, p_1, \ldots, p_d$. Although it would be more accurate to say that A just replaces the information stored in the original nodes, and this view makes more sense when implementing the protocol, it will become convenient later to say that A creates new nodes. The key pairs for the new nodes are sampled as follows. For the leaf node A, user A simply samples a key pair by running $\mathit{\Pi}.\text{Gen}(1^\eta)$. For the remaining nodes, they first sample $s_1 \leftarrow \{0,1\}^{\rho(\eta)}$ and compute the key pair of the first parent p_1 as $\mathit{\Pi}.\text{Gen}(1^\eta, H_{\text{gen}}(s_1))$. For $i \in \{2, \ldots, d\}$ they then compute $s_i \coloneqq H_{\text{dep}}(s_{i-1})$ and set the key pair of p_i to be $\mathit{\Pi}.\text{Gen}(1^\eta, H_{\text{gen}}(s_i))$. The new group key is $k \coloneqq H_{\text{dep}}(s_d)$.

User A only needs to share (encryptions of) the seeds s_i for the other users to update their view of the tree and compute the new group key:

- To share the group key with user B, A computes the ciphertext $c_1 \coloneqq \Pi.\mathrm{Enc}_{pk_B}(s_1)$. B can then compute the seed s_1, then use that to compute the seeds $s_2, \ldots, s_d$, the key pairs of all new nodes on their path to the root and the group key k.
- To share the new group key with users C and D, A computes the ciphertext $c_2 \coloneqq \Pi.\mathrm{Enc}_{pk_X}(s_2)$, where X is the parent of the nodes C and D. Both C and D know the secret key sk_X of their parent and can decrypt c_2.
- To share the new group key with users E, F, G and H, A computes the ciphertext $c_3 \coloneqq \Pi.\mathrm{Enc}_{pk_Y}(s_3)$, where Y is the right child of the root node. Again, all users under Y know sk_Y and can thus decrypt c_3.

The commit c that A shares with all users includes the ciphertexts c_1, c_2 and c_3 and the public keys of all new nodes. Figure 3 illustrates the commit performed by A.

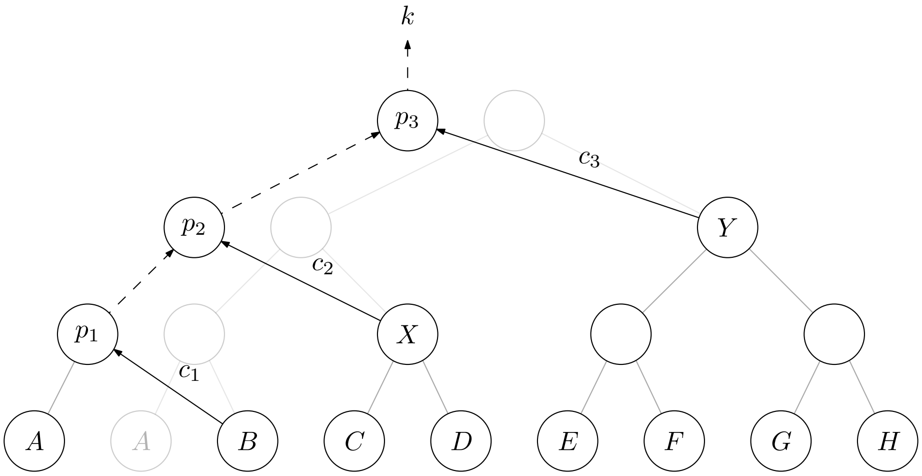


Fig. 3: The commit by user A described in the text. Dashed directed edges illustrate the fact that the target is related to the source via the hash function **Q:** Should I call this a PRF instead? H_{dep}. The solid directed edges illustrate the fact that the seed of the target node is encrypted to the public key of the source node.

The nodes B, X and Y form the *copath* of A: the copath of a node v consists of the sibling of each node on v's path to the root, excluding the root itself. In the ideal case as above, a node performing a commit only has to compute one encryption for each node on its copath, i.e. logarithmically many encryptions in the total number of users.

Remove and update proposals Things look a bit different if the commit contains a remove proposal. Say user A creates a commit that contains a remove proposal for user E. We could just let A replace the nodes on E's direct path E and not encrypt anything for E. However, if A were compromised while replacing E's direct path and performed another commit to update their key material after the compromise, the information leaked in the compromise could still be used to compute the new group key, as it includes the secret keys of the nodes on E's direct path. Instead, E's leaf and all nodes on E's direct path are replaced by *blank* nodes: nodes with no associated key pair. Now A has to encrypt the secret s_3 directly to F and to the parent node of G and H in the commit removing user E. See Figure 4. A blank leaf node can be populated with a new user. A blank node that is not a leaf will be replaced by a non-blank node once some user in the node's subtree performs a commit. Blank nodes are also useful to represent the nodes of a subtree with no users.

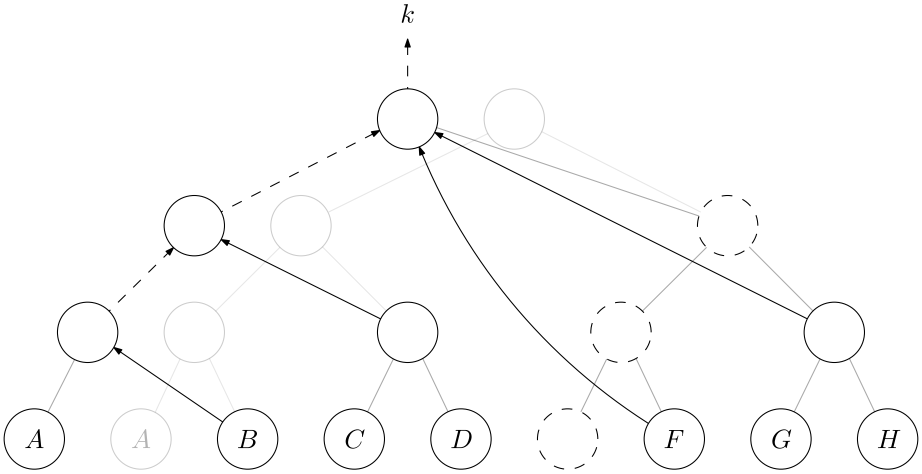


Fig. 4: The commit by user A removing user E described in the text. Nodes with a dashed border represent the (new) blank nodes.

Creating a commit with an update proposal for user a u is analogous. The update proposal simply contains the public key of user u's new leaf, while u stored the corresponding secret key locally when creating the proposal. Because we don't want the committer to know the secret keys along u's direct path, we must again replace these nodes with blank ones and encrypt to the non-blank nodes below directly.

Add proposals Adding a user introduces one new but similar complication. Consider the same group as in Figure 2, but with the leaf of user H blank. Now say user A would like to add user H to the group. Although we would want H

to know all secret keys on their direct path, A can only provide the secrets of their lowest common ancestor, which is the root node in this case. In such a situation where a non-blank node n has a leaf l below it where l does not know n's secret key, we say that l is *unmerged* relative to n. Every non-blank, non-leaf node stores its list of unmerged leaves and whenever one encrypts to a node, one should also encrypt to all its unmerged leaves. A user's leaf becomes "merged" as the nodes on their direct path are replaced and they are provided the seeds to compute the secret keys of the new nodes. Note that for any non-leaf node n, any one of its descendants d and any unmerged leaf of n that is a descendant of d, this leaf must also be an unmerged leaf of d: every commit that replaces d also replaces the node n and if a user at a leaf learns the secret key of the new node for d through its seed, they also learn the seed of and therefore the secret key of the new node for n. Figure 5 shows a commit by user A adding H, followed by another commit by user E.

Resolution We have now seen that when performing a commit, one must pay attention to blank nodes and unmerged leaves when providing encryptions. Instead of only providing encryptions for each node on the copath as in the ideal case, in the general case for each node n on the copath one must provide an encryption for every node in the *resolution* of n. The resolution of a non-blank node is the node itself and the set of all its unmerged leaves. The resolution of a blank leaf is the empty set and the resolution of a blank, non-leaf node is the union of the resolutions of its two children.

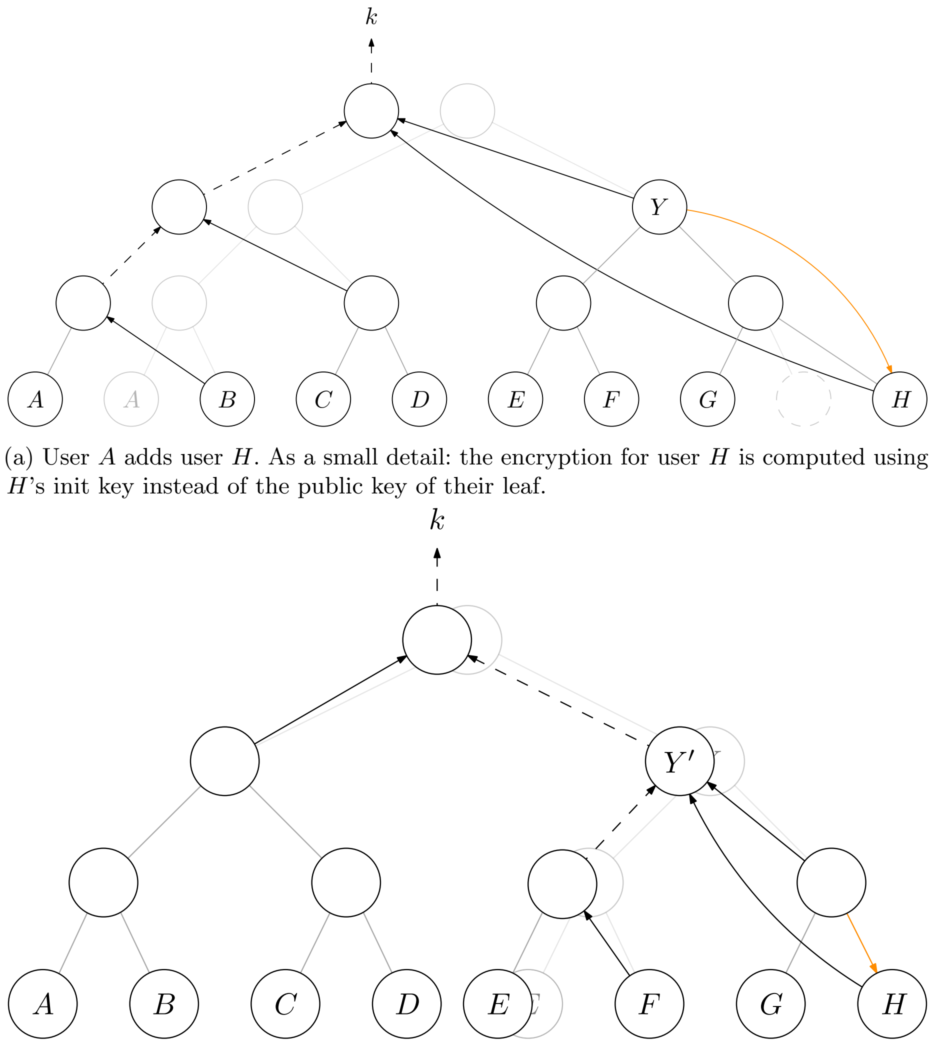
Key packages and welcome messages To encrypt to an existing group member it is clear that we can just use the public key in their leaf. But how do we encrypt to a new user? Before a user joins any group, they publish a *key package*: this contains (among other things) the public key, their so-called *init key*, to be used to encrypt information to the user when they join the group and the public key that should be associated with the user's leaf. The key package is included (or referenced) in the add proposal for the new user. Along with the seed of the committer's and the new user's lowest common ancestor in the tree, the new user must also be given the (public) state of the tree. This information is provided to the new user, encrypted with their init key, by the committer in a *welcome message.*

2 Preliminaries

2.1 Notation

We will use the following notation throughout:

- We write $x \leftarrow S$ to say that x is sampled u.a.r. from the finite set S
- For $n \in \mathbb{N} \setminus \{0\}$, $[n] = \{1, \ldots, n\}$, and for $a, b \in \mathbb{N}$ s.t. $a \leq b$, $[a, b] = \{a, a + 1, \ldots, b\}$
- If $\mathbb{G}$ is a cyclic group of order q and g a generator, then



(a) User A adds user H. As a small detail: the encryption for user H is computed using H's init key instead of the public key of their leaf.

(b) Commit by user E. H is now "merged" relative to node Y'.

Fig. 5: A commit adding user H and another commit by a user E as described in the text. Orange edges illustrate the fact that the target leaf is unmerged relative to the source node. In (a), H is also unmerged relative to Y's right child, but this information is redundant as it follows from H being unmerged relative to Y.

    - We write the group operation in $\mathbb{G}$ multiplicatively
    - h^{-1} denotes the inverse of $h \in \mathbb{G}$
    - $\log_g(h)$ denotes the unique $x \in [q]$ such that $g^x = h$
- We write $b \leftarrow \mathcal{A}$ to denote the event that an adversary $\mathcal{A}$ outputs the bit b when playing a game where it must output a bit in the end
- For $a, b \in \{0,1\}^n$, $a \oplus b$ denotes the XOR of a and b
- We will stick to using κ as the security parameter of private-key encryption schemes and η as the parameter of public-key encryption schemes
- For a function f in the security parameter η (or κ) we will often omit writing η as an argument and simply write f to refer to $f(\eta)$
- For an encryption scheme $\mathit{\Pi}$ that contains an algorithm X, we may refer to $\mathit{\Pi}$'s implementation of X by $\mathit{\Pi}.X$. (E.g. if $\mathit{\Pi}$ is an encryption scheme we can refer to its key-generation algorithm by $\mathit{\Pi}.X$)

2.2 Basic definitions

Our definitions were adapted from [13]. Some simple lemmas and definitions for well-known concepts including private-key and public-key encryption, IND-CPA security, EAV security and the Random Oracle Model (ROM) can be found in section A of the appendix.

Definition 1 (Group-generation algorithm [13, Section 9.3.2]). *A* group-generation algorithm $\mathcal{G}$ *is a probabilistic polynomial-time algorithm that takes as input* 1^η *and outputs* $(\mathbb{G}, q, g)$*, where* $\mathbb{G}$ *is (a description of) a cyclic group with order* q *and* $g \in \mathbb{G}$ *is a generator. A group element is represented as a bit-string of length at most* $\gamma(\eta)$*. We write* $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^\eta)$*.*

Definition 2 (The Decisional Diffie-Hellman (DDH) problem). *Let* $\mathcal{G}$ *a group-generation algorithm. Define the game* $\mathrm{Game}^{\mathrm{DDH}}_{\mathcal{G},\eta}(\mathcal{A})$ *for an adversary* $\mathcal{A}$*:*

1. $\mathcal{G}(1^\eta)$ *is run to obtain* $(\mathbb{G}, q, g)$*, and exponents* $x, y \leftarrow [q]$ *and a bit* $b \leftarrow \{0,1\}$ *are sampled.*
2. *The adversary* $\mathcal{A}$ *is given* $\mathbb{G}$*,* q*,* g*,* $h_1 := g^x, h_2 := g^y$ *and*

$$k = \begin{cases} g^{x \cdot y} & b = 0 \\ \tilde{k} & b = 1 \end{cases}$$

where $\tilde{k} \leftarrow \mathbb{G}$*.*

3. $\mathcal{A}$ *outputs a bit* b'*. The output of the game is defined to be* 1 *if* $b' = b$*, and* 0 *otherwise.*

Definition 3 (Hardness of the DDH problem [13, Definition 9.64]). *The DDH problem is* (t, ε)-hard relative to $\mathcal{G}$ *if for all* η*, for any adversary* $\mathcal{A}$ *running in time* $t(\eta)$ *we have*

$$\mathrm{Adv}^{\mathrm{DDH}}_{\mathcal{G},\eta}(\mathcal{A}) := 2 \cdot \left(\Pr\left[\mathrm{Game}^{\mathrm{DDH}}_{\mathcal{G},\eta}(\mathcal{A}) = 1\right] - \frac{1}{2} \right) \leq \varepsilon(\eta).$$

In the following definition we will refer to "key-derivation functions". This is only meant as a hint to the reader. We do not provide a definition here, as we will always model such a function as a random oracle (see Section A.3).

Definition 4 (DHIES [13, Construction 12.19]). *Let $\mathcal{G}$ a group-generation algorithm. Let Π_s a private-key encryption scheme where $\Pi_s.\mathrm{Gen}(1^\eta)$ samples a key u.a.r. from $\{0,1\}^\eta$. Let $\mathcal{H}_{\mathrm{DH}} = \{H_{\mathrm{DH}}^{(\eta)} \mid \eta \in \mathbb{N}\}$ a family of key-derivation functions where $H_{\mathrm{DH}}^{(\eta)} \colon \{0,1\}^* \to \{0,1\}^\eta$. We write $H_{\mathrm{DH}} \coloneqq H_{\mathrm{DH}}^{(\eta)}$ when η is clear from the context. Define the algorithms* Gen, Enc *and* Dec *as follows:*

- Gen*: on input 1^η run $\mathcal{G}(1^\eta)$ to obtain $(\mathbb{G}, q, g)$. Sample $x \leftarrow [q]$ and set $h_1 \coloneqq g^x$. Set $pk \coloneqq \langle \mathbb{G}, q, g, h_1 \rangle$ and $sk \coloneqq \langle \mathbb{G}, q, g, x \rangle$, and output (pk, sk). The message space is the message space of Π_s.*
- Enc*: on input a public key $\langle \mathbb{G}, q, g, h_1 \rangle$ and a message m, sample $y \leftarrow [q]$, set $h_2 \coloneqq g^y, k \coloneqq H_{\mathrm{DH}}(h_1^y)$*[1]*, compute $c' \leftarrow \Pi_s.\mathrm{Enc}_k(m)$ and output the ciphertext $\langle h_2, c' \rangle$.*
- Dec*: on input a private key $\langle \mathbb{G}, q, g, x, H_{\mathrm{DH}} \rangle$ and a ciphertext $\langle h_2, c' \rangle$, compute $k \coloneqq H(h_2^x)$ and output $\Pi_s.\mathrm{Dec}_k(c')$. If the ciphertext is not of the right form or $\Pi_s.$*Dec *outputs $\perp$, output $\perp$.*

The public-key encryption scheme $\Pi_{\mathrm{DH}} \coloneqq (\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$ is called the Diffie-Hellman Integrated Encryption Scheme (DHIES).

When using the DHIES scheme later on, we will set $pk \coloneqq h$ and $sk \coloneqq x$ in Gen *for simplicity. In practice $\mathbb{G}, q, g$ and H_{DH} will be known.*

Under the DDH assumption (i.e. the assumption that the DDH problem is hard relative to $\mathcal{G}$), using DHIES with an EAV secure private-key scheme gives an IND-CPA secure public-key encryption scheme in the ROM, as proven in [13, Theorem 12.12].

3 TreeKEM security

The following theorem, stated informally here, is our main practical result.

Theorem 1 (Informal). *If the DHIES scheme is used in TreeKEM, the private-key encryption scheme in DHIES is $(t, \varepsilon_{\mathrm{EAV}})$-EAV-secure and the DDH problem is $(t, \varepsilon_{\mathrm{DDH}})$-hard in the Diffie-Hellman group, then for all c, p, u, TreeKEM is $(\tilde{t}, \tilde{\varepsilon}, c, p, u)$-secure in the ROM with $\tilde{t} \approx t$ and*

$$\begin{aligned}
\tilde{\varepsilon} &= 2 \cdot \delta \cdot N \cdot \varepsilon_{\mathrm{EAV}} + 2 \cdot N \cdot \varepsilon_{\mathrm{DDH}} + \frac{2 \cdot m_{\mathrm{DH}} \cdot N^2}{q} + \frac{m_{\mathrm{s}} \cdot N}{2^{\rho-1}} \\
\delta &\coloneqq u \\
N &\coloneqq 2 \cdot c \cdot \log(u) + u + p
\end{aligned}$$

where

[1] Where for $h \in \mathbb{G}$, $H_{\mathrm{DH}}(h)$ denotes the output of H_{DH} with the binary representation of h given as input.

- *c is the number of commits created*
- *p is the number of add or update proposals created*
- *u is the maximum number of users*
- *m_{s} is the number of queries to the functions H_{gen} or H_{dep} in TreeKEM*
- *m_{DH} is the number of queries to H_{DH} in DHIES*
- *q is the size of the Diffie-Hellman group*
- *ρ is the number of bits of randomness used by DHIES for key generation*

In Section C of the appendix we restate the above theorem formally. To this end, we also provide the formal definitions for propose and commit CGKA schemes needed to state our result and a high-level description of how to instantiate (the essence of) the TreeKEM protocol with our definitions.

3.1 Interpreting the result

Let us go through a concrete example to see what level of security this bound guarantees for TreeKEM. We will look at the `MLS_128_DHKEMX25519_AES128GCM_SHA256_Ed25519` cipher suite specified in [6, Section 17.1], which uses Curve25519 as the Diffie-Hellman group, SHA2-256 as the key-derivation function in DHIES, and AES with a 128-bit key size for private-key encryption. We will assume that both Curve25519 and AES have a 128-bit security level, so for example we can set $(t, \varepsilon_{\mathrm{EAV}}) = (t, \varepsilon_{\mathrm{DDH}}) = (2^{80}, 2^{-48})$. The group size for Curve25519 is at least 2^{252} [8]. The seeds sampled in MLS (called "path secrets" in the RFC) and used as arguments to H_{gen} and H_{dep} are 256 bits with SHA2-256 as the key-derivation function [6], meaning that $\rho = 256$. Furthermore we can set $m_{\mathrm{DH}} \leq 2^{60}$ and $m_{\mathrm{s}} \leq 2^{61}$.

For the parameters c and u we can consider a large group of about 1000 users, existing for 5 years and making one commit every hour. Then $u \leq 2^{10}$ and $c \leq 2^{16}$. For p, if we assume that a significant fraction of the users will want to update with every commit and that add proposals are comparatively rare, then we can bound it by $p \leq c \cdot u$.

These numbers yield the following results:

$$\begin{aligned} \delta &\leq 2^{10}, N \leq 2^{27} \text{ (dominated by } p) \\ \tilde{\varepsilon} &\leq 2^{37} \cdot \varepsilon_{\mathrm{EAV}} + 2^{27} \cdot \varepsilon_{\mathrm{DDH}} + \frac{1}{2^{136}} \\ &\leq \frac{1}{2^{10}} \end{aligned}$$

with the $\varepsilon_{\mathrm{EAV}}$ term dominating the result. This gives a security level of $\tilde{t}/\tilde{\varepsilon} \geq 2^{90}$. Since private-key encryption is relatively cheap, using 256-bit AES would have a small impact on the performance and would increase the security level to 100 bits. Finally, using 256-bit AES and a Diffie-Hellman group with a 256-bit security level yields 139 bits of security for TreeKEM.

The previous best result in [1, Theorem 3] proved a bound with

$$\tilde{\varepsilon} \leq 2 \cdot N^2 \cdot \varepsilon + \mathrm{negl}(\eta)$$

for the GSD game, where ε is the IND-CPA security of the underlying public-key encryption scheme. Using the above parameters for a 128-bit security level this implies only 73 bits of security (and the result remains unchanged with 256-bit AES). Therefore, our result significantly improves the guaranteed security level of TreeKEM (in the ROM).

4 Tighter GSD security

In this section we present the main theoretical result behind Theorem 1 in Theorem 2. To this end, we first define a modified version of the public-key GSD game from [1] and then proceed to prove Theorem 2 in detail.

4.1 Seeded GSD with Dependencies

The graph constructed in the public-key GSD game and the tree structure behind the TreeKEM protocol clearly resemble each other. Let $\mathit{\Pi}$ the public-key encryption scheme in use, where $\mathit{\Pi}.\mathrm{Gen}(1^\eta)$ samples $\rho(\eta)$ random bits. We can make some small modifications to the public-key GSD game such that the operations performed in TreeKEM match the ones performed in this modified GSD game. Take the functions $H_{\mathrm{gen}}, H_{\mathrm{dep}}$ used in TreeKEM and first modify the game as follows:

- the key pair of a node v is generated by sampling a seed $s_v \in \{0,1\}^{\rho(\eta)}$ and computing $(pk_v, sk_v) = \mathit{\Pi}.\mathrm{Gen}(1^\eta, s_v)$
- encryption queries encrypt the seed of the target node instead of its secret key

Now the generation of key pairs and the encryptions computed in TreeKEM match what is done in this adapted GSD game. (Although the key pair of a leaf in TreeKEM need not be generated through such a seed, we assumed this to be equivalent to running $\mathit{\Pi}.\mathrm{Gen}(1^\eta)$ in Definition 11.) To model the fact that in TreeKEM a seed of a node may depend on the seed of another node through H_{dep} (as in the new direct path computed in a commit), we introduce a new type of edge which we call a *seed dependency*: a seed dependency (u, v) implies that $s_v = H_{\mathrm{dep}}(s_u)$. We will call our adaptation of the public-key GSD game *Seeded GSD with Dependencies* (SD-GSD).

TODO: Allow adversary to adaptively create nodes and seed dependencies.

- adapt proofs to guess from $[N]$ as opposed to $[n]$
- handle case where Q_{s} triggered because adversary queried a seed and that seed later became the seed of a newly created safe node

The definition provided here is inspired by the definition of the public-key GSD game (Definition 7) and the proof of Theorem 3 in [1]. A very similar definition appears in [3], providing essentially the same abstraction over TreeKEM and also allowing for an adversary to provide the randmoness used for encryption and key generation. However, our definition has one notable difference: when asking to be challenged on a node with seed s, the adversary must distinguish $H_{\text{dep}}(s)$ from random as opposed to s. This stays true to how the group key is computed in TreeKEM and has the significant advantage of greatly simplifying our proof. On the other hand, the security implied by our definition is weaker (at least in the ROM), as it only guarantees that an adversary cannot compute the seed of the challenge node, whereas the other definitions guarantee that this seed cannot be distinguished from random.

Definition 5 (The SD-GSD game). *Let $\mathit{\Pi} = (\text{Gen}, \text{Enc}, \text{Dec})$ a public-key encryption scheme, where $\text{Gen}(1^\eta)$ uses $\rho(\eta)$ bits of randomness and $\{0,1\}^{\rho(\eta)}$ is a subset of the message space. Let $\mathcal{H}_{\text{gen}} = \{H_{\text{gen}}^{(\eta)} \mid \eta \in \mathbb{N}\}, \mathcal{H}_{\text{dep}} = \{H_{\text{DH}}^{(\eta)} \mid \eta \in \mathbb{N}\}$ families of functions with $H_{\text{gen}}^{(\eta)}, H_{\text{dep}}^{(\eta)} \colon \{0,1\}^{\rho(\eta)} \to \{0,1\}^{\rho(\eta)}$. We will write $H_{\text{gen}} \coloneqq H_{\text{gen}}^{(\eta)}, H_{\text{dep}} \coloneqq H_{\text{dep}}^{(\eta)}$ and $\rho \coloneqq \rho(\eta)$ if η is clear from the context. Define the game $\text{Game}^{\text{SD-GSD}}_{(\mathit{\Pi},\mathcal{H}_{\text{gen}},\mathcal{H}_{\text{dep}}),\eta}(\mathcal{A})$ for an adversary $\mathcal{A}$:*

1. *The adversary $\mathcal{A}$ and outputs $n \in \mathbb{N}$ and a list of dependencies $D = \{(a_i, b_i)\}_{i=1}^m \in [n]^2$. For each $v \in [n]$:*
   (i) – ***Case** $v = b_i$ **for some** i **(v is the target of some dependency):*** *set $s_v = H_{\text{dep}}(s_{a_i})$.*
   – ***Otherwise:*** *sample $s_v \leftarrow \{0,1\}^\rho$.*
   We call s_v the seed *of the node v and a tuple $(a, b) \in D$ a* seed dependency*.*
   (ii) Compute $(pk_v, sk_v) = \text{Gen}(1^\eta, H_{\text{gen}}(s_v))$.
   Set $\mathcal{C} = E = \varnothing$. We call the directed graph $([n], E)$ a GSD graph *of* size *n.*
2. *$\mathcal{A}$ may adaptively do the following queries:*
   - $\text{reveal}(v)$ *for $v \in [n]$: $\mathcal{A}$ is given pk_v.*
   - $\text{encrypt}(u, v)$ *for $u, v \in [n], u \neq v, (u, v) \notin E$: (u, v) is added to E and $\mathcal{A}$ is given $c \leftarrow \text{Enc}_{pk_u}(s_v)$.*
   - $\text{corrupt}(v)$ *for $v \in [n], v \notin \mathcal{C}$: $\mathcal{A}$ is given s_v and v is added to $\mathcal{C}$. We call such a node $v \in \mathcal{C}$* corrupted*. All nodes not reachable from any corrupted node in the graph $([n], E \cup D)$ are* safe *(while all other nodes are* unsafe*) and we call their seeds* hidden *(even if an unsafe node happens to have the same seed).*
3. *$\mathcal{A}$ outputs a node $v \in [n]$. We call v the* challenge node*. A bit $b \leftarrow \{0,1\}$ is sampled and $\mathcal{A}$ is given*

$$r = \begin{cases} H_{\text{dep}}(s_v) & b = 0 \\ s & b = 1 \end{cases},$$

   where $s \leftarrow \{0,1\}^\rho$.[2] *$\mathcal{A}$ may continue to do queries as before.*
4. *$\mathcal{A}$ outputs a bit b'. The output of the game is defined to be 1 if $b' = b$, and 0 otherwise.*

We require an adversary playing the above game to adhere to the following:

[2] Note that (in general) r is not a hidden seed, as (with overwhelming probability) it is not the seed of any node.

- *The challenge node is safe*
- *The graph $(V, E \cup D)$ always remain acyclic and without self-loops*
- *All paths in the graph (V, D) are vertex disjoint*[3]

It is interesting to note that previous definitions of the GSD game also included the following restrictions to the adversary:

- The challenge node always remains a sink
- reveal is never queried on the challenge node

Our proof in the ROM does not require these restrictions. If H_{dep} is modelled as a random oracle, then an encryption edge or seed dependency outgoing from the challenge node, or knowing its public key gives no advantage to $\mathcal{A}$, as by the assumption of H_{dep} being a random oracle the only way to learn information about $H_{\text{dep}}(s)$ is by querying s.

Definition 6 (SD-GSD security). *Let $\mathit{\Pi}, \mathcal{H}_{\text{gen}}$ and $\mathcal{H}_{\text{dep}}$ as in Definition 5 above and let $t, \varepsilon, N, \delta$ functions in η. The triple $(\mathit{\Pi}, H_{\text{gen}}, H_{\text{dep}})$ is $(t, \varepsilon, N, \delta)$-*SD-GSD-secure *if for all η, for any adversary $\mathcal{A}$ constructing a GSD graph of size at most $N(\eta)$ and indegree at most $\delta(\eta)$ and running time in $t(\eta)$ we have*

$$\mathrm{Adv}^{\mathrm{SD-GSD}}_{(\mathit{\Pi}, \mathcal{H}_{\text{gen}}, \mathcal{H}_{\text{dep}}), \eta}(\mathcal{A}) := 2 \cdot \left(\Pr\left[\mathrm{Game}^{\mathrm{SD-GSD}}_{(\mathit{\Pi}, \mathcal{H}_{\text{gen}}, \mathcal{H}_{\text{dep}}), \eta}(\mathcal{A}) = 1 \right] - \frac{1}{2} \right) \leq \varepsilon(\eta).$$

Since in this work we are interested in SD-GSD security for the case where H_{gen} and H_{dep} are modelled as random oracles and our focus is on the encryption scheme being used, we introduce the following definition for convenience.

Definition 7 (SD-GSD security in the ROM). *A public-key encryption scheme $\mathit{\Pi}$ is $(t, \varepsilon, N, \delta)$-*SD-GSD-secure in the ROM *if the triple $(\mathit{\Pi}, \mathcal{H}_{\text{gen}}, \mathcal{H}_{\text{dep}})$ is $(t, \varepsilon, N, \delta)$-SD-GSD-secure when H_{gen} and H_{dep} are modelled as random oracles. For security parameter η and an adversary $\mathcal{A}$, we write* $\mathrm{Game}^{\mathrm{SD-GSD}}_{\mathit{\Pi}, \eta}(\mathcal{A})$ *to denote the game where H_{gen} and H_{dep} are modelled as random oracles and* $\mathrm{Adv}^{\mathrm{SD-GSD}}_{\mathit{\Pi}, \eta}(\mathcal{A})$ *for $\mathcal{A}$'s advantage in this game.*

4.2 Proving SD-GSD security for DHIES in the ROM

Theorem 2. *Let $\mathit{\Pi}_{\text{DH}}$ denote the DHIES scheme instantiated with a group-generation algorithm $\mathcal{G}$ and a private-key encryption scheme $\mathit{\Pi}_s$. If $\mathit{\Pi}_s$ is $(t, \varepsilon_{\text{EAV}})$-EAV-secure, the DDH problem is $(t, \varepsilon_{\text{DDH}})$-hard relative to $\mathcal{G}$ and the function H_{DH} in $\mathit{\Pi}_{\text{DH}}$ is modelled as a random oracle, then for any δ, N with $\delta \leq N$, $\mathit{\Pi}_{\text{DH}}$ is $(\tilde{t}, \tilde{\varepsilon}, N, \delta)$-SD-GSD-secure in the ROM with*[4]

$$\tilde{\varepsilon} = 2 \cdot \delta \cdot N \cdot \varepsilon_{\text{EAV}} + 2 \cdot N \cdot \varepsilon_{\text{DDH}} + \frac{2 \cdot m_{\text{DH}} \cdot N^2}{q} + \frac{m_{\text{s}} \cdot N}{2^{\rho - 1}},$$

[3] This ensures that any node is only the target of at most one seed dependency, such that the computation of seeds is well-defined.

[4] Note that in the following equality we have omitted writing the argument η to the various functions and are implying that the equality holds for all η.

where m_{s} is an upper bound on the number of queries made to either H_{gen} or H_{dep}, m_{DH} is an upper bound on the number of queries made to H_{DH}, q is a lower bound on the size of the group output by $\mathcal{G}$ and ρ is the number of random bits sampled by $\mathit{\Pi}_{\mathrm{DH}}.\mathrm{Gen}$, and with

$$\begin{aligned}\tilde{t} = t - \mathcal{O}\big(&\rho \cdot t_{\mathrm{sample}} \cdot m_{\mathrm{s}} + (\gamma + \eta \cdot t_{\mathrm{sample}}) \cdot m_{\mathrm{DH}} \\ &+ N \cdot ((\rho + \eta) \cdot t_{\mathrm{sample}} + m_{\mathrm{DH}} \cdot t_{\mathrm{op}} + t_{\mathit{\Pi}_{\mathrm{DH}}.\mathrm{Gen}}) \\ &+ N^2 \cdot t_{\mathit{\Pi}_{\mathrm{DH}}.\mathrm{Enc}}\big),\end{aligned}$$

where the various variables denote the following

- t_{sample}*: time to sample a uniform bit*
- $t_{\mathit{\Pi}_{\mathrm{DH}}.\mathrm{Enc}}$*: time to encrypt $s \in \{0,1\}^{\rho}$ with $\mathit{\Pi}_{\mathrm{DH}}$*
- $t_{\mathit{\Pi}_{\mathrm{DH}}.\mathrm{Gen}}$*: runtime of $\mathit{\Pi}_{\mathrm{DH}}.\mathrm{Gen}(1^{\eta})$ (which is strictly greater than the runtime of $\mathit{\Pi}_{\mathrm{DH}}.\mathrm{Gen}(1^{\eta}, r)$ for input randomness r)*
- t_{op}*: time to perform the group operation in a group output by $\mathcal{G}(1^{\eta})$*
- γ*: maximum length of any query to H_{DH}*

In contrast, the result in [1] achieves a security loss in $\mathcal{O}(N^2)$ and reduces to the IND-CPA security of the public-key encryption scheme.

For ease of exposition, we will assume that $\mathcal{G}(1^{\eta})$ is deterministic, as is the case in practice. We will therefore set $pk := h_1, sk := x$ in $\mathit{\Pi}_{\mathrm{DH}}.\mathrm{Gen}$, as $\mathbb{G}, q, g$ are implied by η. The results nevertheless hold also for the general case.

Intuition Consider an arbitrary SD-GSD adversary $\mathcal{A}$. For an execution of $\mathrm{Game}^{\mathrm{SD-GSD}}_{\mathit{\Pi}_{\mathrm{DH}},\eta}(\mathcal{A})$ we say "$\mathcal{A}$ *wins*" to denote the event $\mathrm{Game}^{\mathrm{SD-GSD}}_{\mathit{\Pi}_{\mathrm{DH}},\eta}(\mathcal{A}) = 1$. As usual with random oracles we proceed by a case distinction on whether they were queried on some interesting value. Accordingly, let Q_x denote the event that $\mathcal{A}$ queries H_x on a hidden seed for $x \in \{\mathrm{gen}, \mathrm{dep}\}$. Then we can write

$$\begin{aligned}\Pr[\mathcal{A}\text{ wins}] &= \Pr[\mathcal{A}\text{ wins} \wedge Q_{\mathrm{dep}}] + \Pr\big[\mathcal{A}\text{ wins} \wedge \overline{Q_{\mathrm{dep}}}\,\big] \\ &\leq \Pr[\mathcal{A}\text{ wins} \wedge Q_{\mathrm{dep}}] + \Pr\big[\mathcal{A}\text{ wins} \mid \overline{Q_{\mathrm{dep}}}\,\big] \\ &\overset{(\dagger)}{=} \Pr[\mathcal{A}\text{ wins} \wedge Q_{\mathrm{dep}}] + \frac{1}{2} \\ &\leq \Pr[Q_{\mathrm{dep}}] + \frac{1}{2} \\ &\leq \Pr[Q_{\mathrm{s}}] + \frac{1}{2},\end{aligned} \tag{1}$$

where $Q_{\mathrm{s}} := Q_{\mathrm{gen}} \cup Q_{\mathrm{dep}}$ (s for *seed*). Step (†) intuitively holds because without having queried H_{dep} for any hidden seed, in particular the seed s_v of the challenge node v, $H_{\mathrm{dep}}(s_v)$ is a uniformly random value from $\mathcal{A}$'s perspective. Therefore, it can do no better than guessing to distinguish $H_{\mathrm{dep}}(s_v)$ from $s \leftarrow \{0,1\}^{\rho}$.

TODO: Motivate why we introduce Q_{s}. (Reason: If we try to bound Q_{dep} by itself, we must separately deal with the case where the adversary was able to trigger it at a node v by triggering Q_{gen} at a parent node p and subsequently decrypting a ciphertext. But our argument To eliminate this, we want to look at the point in time where either of the two events was first triggered.)

The heart of the proof is to bound $\Pr[Q_\mathrm{s}]$. When the adversary first triggers Q_s by querying the seed of some safe node w, (with overwhelming probability w will be the only node with this seed and) it can only have learned the seed through encryptions $c_1 \leftarrow \mathit{\Pi}_\mathrm{DH}.\mathrm{Enc}_{pk_{u_1}}(s_w), \ldots, c_d \leftarrow \mathit{\Pi}_\mathrm{DH}.\mathrm{Enc}_{pk_{u_d}}(s_w)$ where $(u_1, w), \ldots, (u_d, w)$ are edges in the GSD graph (obtained through corresponding queries $\mathrm{encrypt}(u_1, w), \ldots, \mathrm{encrypt}(u_d, w)$). The only other potential source of information about s_w would be a seed dependency (p, w), but this tells $\mathcal{A}$ nothing: Since w is safe, p would also be safe and $H_\mathrm{dep}(s_p)$ cannot have been queried due to the assumption that w was the first node to trigger Q_s. Without having queried $H_\mathrm{dep}(s_p)$, by virtue of H_dep being a random oracle s_w has the same distribution as a seed without a dependency from $\mathcal{A}$'s perspective (uniformly random). See Figure 6 for an illustration of node w in the GSD graph.

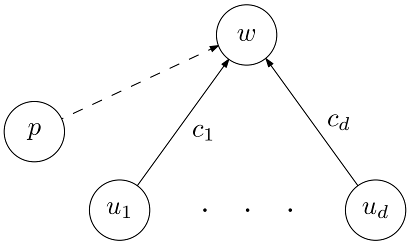


Fig. 6: Illustration of the GSD graph when Q_s is triggered at a node w. The dashed edge represents a seed dependencey (p, w) and the remaining edges represent encryption queries $c_i \leftarrow \mathrm{encrypt}(u_i, w)$.

The proof in [1] simply argued that this is not too likely if these encryptions were made with an IND-CPA secure scheme. In the context of the DHIES scheme we can say more about these encryptions and achieve a better reduction loss. Let $(\mathbb{G}, q, g) = \mathcal{G}(1^\eta)$. Let $x_i = \log_g(pk_{u_i})$. Each encryption c_i is a tuple of the form $\langle g^{y_i}, \mathit{\Pi}_s.\mathrm{Enc}_{k_i}(s_w)\rangle$ where $y_i \leftarrow [q], k_i = H_\mathrm{DH}(g^{x_i \cdot y_i})$. Now we can again do a case distinction on whether H_DH was queried for (the encoding of) some group element $g^{x_j \cdot y_j}$ or not:

(i) If such a query was made, then $\mathcal{A}$ solved the Diffie-Hellman challenge (g^{x_j}, g^{y_j}). (Remember that we assumed that w is the first node for which Q_s is triggered and as before if w is safe, then so are the nodes u_i. Thus the adversary has not learned the exponent x_i through querying $H_\mathrm{gen}(s_{u_i})$ for any i.)

(ii) If no such query was made, then from $\mathcal{A}$'s perspective all the k_i are independent, uniformly random keys and it still was able to learn s_w from the EAV secure encryptions $\mathit{\Pi}_s.\mathrm{Enc}_{k_1}(s_w), \ldots, \mathit{\Pi}_s.\mathrm{Enc}_{k_d}(s_w)$.

We can bound the probability of either of these events occurring using hardness of the DDH problem relative to $\mathcal{G}$ and EAV security of Π_s, respectively.

To this end, we call a group element $h \in \mathbb{G}$ a *hidden Diffie-Hellman key* if $h = pk_u^{y_{u,v}}$, where (u, v) is an edge in the GSD graph, u is safe and $y_{u,v}$ is the exponent chosen in the DHIES encryption of s_v (i.e. $\mathcal{A}$ was given a ciphertext of the form $\langle g^{y_{u,v}}, \ldots \rangle$ when it queried encrypt(u, v)). Now analogously to above let Q_{DH} the event that $\mathcal{A}$ queries H_{DH} on a hidden Diffie-Hellman key, and let F_{DH} the event that $\mathcal{A}$ triggers Q_{DH} when Q_{s} has not (yet) been triggered. **TODO:** Querying a hidden DH key can also happen when the adversary first queries the group element and then later does an encrypt query that creates an edge with that group element as the key. Add notes in SD-GSD adversaries in proofs to explain how they keep track of this and check if any arguments must be adapted (what is essential is that the adversary has not queried $H_{\mathrm{DH}}(k)$ for any hidden key k and the simulations are thus distributed identically). Then we can split the event Q_{s} into two cases as motivated above:

$$\Pr[Q_{\mathrm{s}}] = \Pr[Q_{\mathrm{s}} \wedge F_{\mathrm{DH}}] + \Pr\left[Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}\,\right].$$

We bound $\Pr[Q_{\mathrm{s}} \wedge F_{\mathrm{DH}}]$ and $\Pr\left[Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}\,\right]$ in Lemma 3 and Lemma 2, respectively. Overall this gives us a bound on the advantage of $\mathcal{A}$ using (1). (To be precise, the event $Q_{\mathrm{s}} \wedge F_{\mathrm{DH}}$ is a superset of case ((i)) above. However, the argument applied in Lemma 3 gives the same bound for either event and this more general event has the advantage of being simpler.)

Proof (of Theorem 2). Let δ, N functions in η (mapping to $\mathbb{N}$) with $\delta \leq N$. Let η arbitrary and let $\mathcal{A}$ an arbitrary SD-GSD adversary constructing a GSD graph of size at most $N(\eta)$ and indegree at most $\delta(\eta)$, making at most $m_{\mathrm{s}}(\eta)$ queries to H_{gen} or H_{dep} and at most $m_{\mathrm{DH}}(\eta)$ queries to H_{DH}, each of length at most $\gamma(\eta)$, and running in time $\tilde{t}(\eta)$. We will use the events defined above.

We first justify step (†) in (1). Note that by the rules imposed on the adversary in the SD-GSD game, the challenge node v is safe and its seed s_v thus indeed hidden. If Q_{dep} does not hold, then $\mathcal{A}$ has not queried H_{dep} for s_v and, by virtue of H_{dep} being a random oracle, $H_{\mathrm{dep}}(s_v)$ is a uniformly distributed value in $\{0,1\}^\rho$ from $\mathcal{A}$'s perspective. The value s follows the same distribution. Thus, $\mathcal{A}$ behaves the same when given either $r = s$ or $r = H_{\mathrm{dep}}(s_v)$ and

$$\begin{aligned} \Pr\left[1 \leftarrow \mathcal{A} \mid \overline{Q_{\mathrm{dep}}}, b = 1\right] &= \Pr\left[1 \leftarrow \mathcal{A} \mid \overline{Q_{\mathrm{dep}}}, r = s\right] \\ &= \Pr\left[1 \leftarrow \mathcal{A} \mid \overline{Q_{\mathrm{dep}}}, r = H_{\mathrm{dep}}(s_v)\right] \\ &= \Pr\left[1 \leftarrow \mathcal{A} \mid \overline{Q_{\mathrm{dep}}}, b = 0\right]. \end{aligned} \tag{2}$$

Therefore

$$\begin{aligned}\Pr\big[\mathcal{A} \text{ wins} \mid \overline{Q_{\text{dep}}}\,\big] &= \Pr\big[1 \leftarrow \mathcal{A} \mid \overline{Q_{\text{dep}}}, b=1\big] \cdot \frac{1}{2} \\ &\quad + \Pr\big[0 \leftarrow \mathcal{A} \mid \overline{Q_{\text{dep}}}, b=0\big] \cdot \frac{1}{2} \\ &\overset{(2)}{=} \Pr\big[1 \leftarrow \mathcal{A} \mid \overline{Q_{\text{dep}}}, b=0\big] \cdot \frac{1}{2} \\ &\quad + \Pr\big[0 \leftarrow \mathcal{A} \mid \overline{Q_{\text{dep}}}, b=0\big] \cdot \frac{1}{2} \\ &= \frac{1}{2}.\end{aligned}$$

By Lemma 3 on page 23 we have[5]

$$\Pr[Q_{\text{s}} \wedge F_{\text{DH}}] \leq N \cdot \varepsilon_{\text{DDH}} + \frac{m_{\text{DH}} \cdot N^2}{q}.$$

and by Lemma 2 on page 22 we have

$$\Pr\big[Q_{\text{s}} \wedge \overline{F_{\text{DH}}}\,\big] \leq \delta \cdot N \cdot \varepsilon_{\text{EAV}} + \frac{m_{\text{s}} \cdot N}{2^{\rho}},$$

so we know that

$$\Pr[Q_{\text{s}}] \leq N \cdot \varepsilon_{\text{DDH}} + \delta \cdot N \cdot \varepsilon_{\text{EAV}} + \frac{m_{\text{DH}} \cdot N^2}{q} + \frac{m_{\text{s}} \cdot N}{2^{\rho}} = \tilde{\varepsilon}(\eta)/2. \tag{3}$$

Then

$$\begin{aligned}\text{Adv}^{\text{SD}-\text{GSD}}_{\mathit{\Pi},\eta}(\mathcal{A}) &= 2 \cdot \left(\Pr[\mathcal{A} \text{ wins}] - \frac{1}{2}\right) \\ &\overset{(1)}{\leq} 2 \cdot \Pr[Q_{\text{s}}] \\ &\overset{(3)}{\leq} \tilde{\varepsilon}(\eta).\end{aligned}$$

Reducing to EAV security Recall case (ii) in the high-level discussion of Theorem 2: the adversary $\mathcal{A}$ was able to learn the seed s_w given the EAV secure encryptions $\mathit{\Pi}_s.\text{Enc}_{k_1}(s_w), \ldots, \mathit{\Pi}_s.\text{Enc}_{k_d}(s_w)$. We can see $\mathcal{A}$ as an adversary against a security game where $\mathcal{A}$ is given d EAV secure encryptions $c_1 \leftarrow \mathit{\Pi}_s.\text{Enc}_{k_1}(m), \ldots, c_d \leftarrow \mathit{\Pi}_s.\text{Enc}_{k_d}(m)$ of a message m with $k_i \leftarrow \mathit{\Pi}_s.\text{Gen}(1^{\eta})$ and must compute m. If we can prove that beating such a game is hard, then we can bound the probability of $\mathcal{A}$ actually learning s_w in this way.

This is exactly how we proceed in this section. Instead of asking the adversary to compute an encrypted message m, we turn to a more familiar decisional

[5] Note that we are again omitting the argument η from the functions on the right hand side ($N, \varepsilon_{\text{DDH}}$ and m_{DH} in this case).

formulation as in the IND-CPA game (where the adversary may choose a pair m_0, m_1 and must distinguish whether the d ciphertexts encrypt m_0 or m_1). We call this security notion *EAV security under multiple (M) independent (I) encryptions of a single (S) pair of messages* (MIS-EAV).

Definition 8 (The MIS-EAV game). *Let κ denote the security parameter and let Π a private-key encryption scheme. Define the game* $\mathrm{Game}^{\mathrm{MIS-EAV}}_{\Pi,\kappa}(\mathcal{A})$ *for an adversary $\mathcal{A}$:*

1. *The adversary $\mathcal{A}$ outputs $q \in \mathbb{N}$ and a pair of messages m_0, m_1 of the same length. We refer to q as the number of* queries *made by $\mathcal{A}$.*
2. *A bit $b \leftarrow \{0,1\}$ is sampled. For each $i \in [q]$, $\mathcal{A}$ is given an encryption $c_i \leftarrow \Pi.\mathrm{Enc}_{k_i}(m_b)$ where $k_i \leftarrow \Pi.\mathrm{Gen}(1^\kappa)$ is generated independently of the other keys.*
3. *$\mathcal{A}$ outputs a bit b'. The output of the game is defined to be 1 if $b' = b$, and 0 otherwise.*

Definition 9 (MIS-EAV security). *A private-key encryption scheme Π is* (t, ε, q)-MIS-EAV-secure *if for all κ, for any adversary $\mathcal{A}$ making at most $q(\kappa)$ queries and running in time $t(\kappa)$ we have*

$$\mathrm{Adv}^{\mathrm{MIS-EAV}}_{\Pi,\kappa}(\mathcal{A}) := 2 \cdot \left(\Pr\left[\mathrm{Game}^{\mathrm{MIS-EAV}}_{\Pi,\kappa}(\mathcal{A}) = 1\right] - \frac{1}{2}\right) \leq \varepsilon(\kappa).$$

Similar to how IND-CPA security for a single encryption query implies IND-CPA security for q queries with a security loss of q by a standard hybrid argument, one can show that EAV security implies MIS-EAV security with the same loss. To see why, recall the hybrid argument for IND-CPA security (as discussed in e.g. [13, Theorem 12.6]): We define the sequence of hybrid games $H_0, \ldots, H_q$ where in the game H_i the first i encryption queries encrypt the second message and the remaining $q - i$ queries encrypt the first message. Then given an IND-CPA adversary $\mathcal{A}$ for multiple encryptions, an IND-CPA adversary $\mathcal{A}'$ is constructed to bound

$$|\Pr[\mathcal{A} \text{ outputs } 0 \text{ in game } H_{i-1}] - \Pr[\mathcal{A} \text{ outputs } 0 \text{ in game } H_i]|$$

for arbitrary i. The adversary $\mathcal{A}'$ simulates H_{i-1} or H_i to $\mathcal{A}$ depending on whether the ciphertext received from the (single-query) IND-CPA challenger, which gets passed on as the response to the i-th query, encrypts the first or the second message from the i-th pair of messages. $\mathcal{A}'$ then uses the encryption oracle to pass on the right encryptions to $\mathcal{A}$ for all other queries. Now notice that if we wanted to simulate to an MIS-EAV adversary we wouldn't need access to an encryption oracle since for the MIS-EAV security game all the other encryptions can easily be generated by $\mathcal{A}'$ sampling the new keys itself.

The argument would of course also work without restricting the adversary to a single pair of messages (which we could call MI-EAV security). However, we will make use of this restriction to provide a tighter reduction for a certain class of schemes in the appendix.

Lemma 1. *Let $\mathit{\Pi}$ a private-key encryption scheme with finite message space. Let $t_{\mathrm{Gen}}, t_{\mathrm{Enc}}$ functions in κ that upper bound the runtime of $\mathit{\Pi}.\mathrm{Gen}$ and $\mathit{\Pi}.\mathrm{Enc}$, respectively. If $\mathit{\Pi}$ is (t, ε)-EAV-secure, then for any function q, $\mathit{\Pi}$ is $(\tilde{t}, q \cdot \varepsilon, q)$-MIS-EAV-secure with $\tilde{t} = t - \mathcal{O}(q \cdot (t_{\mathrm{Gen}} + t_{\mathrm{Enc}}))$.*

The details of the proof can be found in Section B.1 of the appendix.

Lemma 2. *Recall the assumptions, variables and events from the statement and proof of Theorem 2. In particular, assume that $\mathit{\Pi}_s$ is $(t, \varepsilon_{\mathrm{EAV}})$-EAV-secure. Let η arbitrary and let $\mathcal{A}$ an SD-GSD adversary constructing a GSD graph of size at most $N(\eta)$ and indegree at most $\delta(\eta)$, making at most $m_{\mathrm{s}}(\eta)$ queries to H_{gen} or H_{dep} and at most $m_{\mathrm{DH}}(\eta)$ queries to H_{DH}, and running in time $\tilde{t}(\eta)$. Then*

$$\Pr\left[Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}\,\right] \leq \delta \cdot N \cdot \varepsilon_{\mathrm{EAV}} + \frac{m_{\mathrm{s}} \cdot N}{2^{\rho}}.$$

Intuition By Lemma 1 we know that $\mathit{\Pi}_s$ is MIS-EAV secure. Continuing the high-level argument before the proof of Theorem 2, consider the first moment that $\mathcal{A}$ triggers $Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}$ by querying the seed of some safe node w. As intended, it follows from the definition of the event F_{DH} that from $\mathcal{A}$'s perspective all DHIES ciphertexts it got from queries $\mathrm{encrypt}(u, w)$ for any u contain encryptions of s_w under independent, uniformly random keys using $\mathit{\Pi}_s$. Moreover, as already argued once, $\mathcal{A}$ has learned nothing from a potential seed dependency (p, w), so these encryptions are everything $\mathcal{A}$ had at its proposal to learn s_w.

TODO: Add plot.

We can use $\mathcal{A}$'s ability to compute the seed s_w of a safe node w from encryptions of s_w to construct an MIS-EAV adversary: We first guess a node w whose seed $\mathcal{A}$ may query first. Next we give the MIS-EAV challenger s_w and some other independent seed s. We simulate the SD-GSD game to $\mathcal{A}$ and embed the encryptions from the MIS-EAV challenger when answering queries of the form $\mathrm{encrypt}(u, w)$ for any u. Now consider the behavior of $\mathcal{A}$ depending on which seed the challenger chooses to encrypt:

- If the challenger chooses to encrypt s_w, then $\mathcal{A}$ will trigger the event $Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}$ with the same probability as before. We can detect whether $Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}$ gets triggered since all seeds in the simulation are known. If $Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}$ occurs and we guessed w correctly, the event will be triggered at w and $\mathcal{A}$ will query s_w, telling us that the challenger encrypted s_w.
- If the challenger chooses to encrypt s, then $\mathcal{A}$ receives no information about s_w and has negligible probability of querying it.

Thus, the advantage of the MIS-EAV adversary is about $\Pr\left[Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}\,\right]/N$, where the factor $1/N$ arises from guessing w, and using that $\mathit{\Pi}_s$ is MIS-EAV secure we can bound this probability. Since we are only interested in checking whether the event was triggered for w, the adversary can abort when this is no longer possible (w is corrupted, some other hidden seed is queried, etc.). The details of the proof can be found in Section B.2 of the appendix.

For a certain class of schemes, we can improve on Lemma 1 and achieve a tight reduction. This allows use to get rid of the factor δ in Lemma 2. However, we do not use this in our main result and refer the interested reader to Section B.3 of the appendix.

Reducing to the DDH problem

Lemma 3. *Recall the assumptions, variables and events from the statement and proof of Theorem 2. In particular, assume that the DDH problem is $(t, \varepsilon_{\mathrm{DDH}})$-hard relative to $\mathcal{G}$. Let η arbitrary and let $\mathcal{A}$ an SD-GSD adversary constructing a GSD graph of size at most $N(\eta)$ and indegree at most $\delta(\eta)$, making at most $m_{\mathrm{s}}(\eta)$ queries to H_{gen} or H_{dep} and at most $m_{\mathrm{DH}}(\eta)$ queries to H_{DH}, and running in time $\tilde{t}(\eta)$. Then*

$$\Pr[Q_{\mathrm{s}} \wedge F_{\mathrm{DH}}] \leq N \cdot \varepsilon_{\mathrm{DDH}} + \frac{m_{\mathrm{DH}} \cdot N^2}{q}.$$

Intuition We will bound the simpler event F_{DH}. This event tells us that there is some safe node a in the GSD graph with encryption edges to nodes $u_1, \ldots, u_d$, where the query encrypt(a, u_i) returned the ciphertext $\langle g^{y_i}, \mathit{\Pi}_s.\mathrm{Enc}_{k_i}(s_{u_i})\rangle$ with $k_i = H_{\mathrm{DH}}(g^{sk_a \cdot y_i})$, such that $g^{sk_a \cdot y_j}$ was the first hidden Diffie-Hellman key queried by $\mathcal{A}$ for some j. Moreover, at the time $g^{sk_a \cdot y_j}$ was queried, no hidden seed had yet been queried by $\mathcal{A}$, implying that $\mathcal{A}$ had not queried $H_{\mathrm{gen}}(s_a)$ and thus had no information about sk_a besides pk_a (recall that $(pk_a, sk_a) = \mathit{\Pi}_{\mathrm{DH}}.\mathrm{Gen}(1^\eta, H_{\mathrm{gen}}(s_a))$).

TODO: Add plot.

It is interesting to note that our approach does not require that $\mathcal{A}$ has not queried H_{dep} for a hidden seed (i.e. that Q_{dep} was not triggered) as is implied by the event F_{DH}, because knowing $H_{\mathrm{gen}}(s_a)$ is the only way to learn about sk_a. Regardless, we still want to have our definition of F_{DH} include this information, as the bound on $\Pr\left[Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}\right]$ in Lemma 2 on the preceding page relies on the fact that in the event of $Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}$ happening, Q_{DH} was not yet triggered when the event Q_{s} was triggered, i.e. when either the event Q_{gen} *or* the event Q_{dep} was triggered.

The intuition is clear that this means that $\mathcal{A}$ solved the Diffie-Hellman challenge (g^{sk_a}, g^{y_j}). What is not immediately clear is how to embed a *given* Diffie-Hellman challenge (g^x, g^y) from an instance of the DDH game and use $\mathcal{A}$ to tell whether the key k chosen by the challenger is the real key $g^{x \cdot y}$ or a uniformly random group element. An intuitive strategy would be to embed the challenge by setting $pk_a = g^x$ and $g^{y_j} = g^y$, which involves guessing u_j, and simply checking whether for any of the queries q_i to H_{DH} by $\mathcal{A}$, such that q_i encodes a group element in $\mathbb{G}$, it holds that $q_i = k$. Now:

- If $k = g^{x \cdot y}$, $\mathcal{A}$ triggers F_{DH} and we guessed a and u_j correctly, then indeed as described above $q_i = g^{sk_a \cdot y_j} = g^{x \cdot y} = k$ will hold for some i.
- If k is a random group element, then $\mathcal{A}$ has negligible probability of querying k, as no information about k is ever leaked to $\mathcal{A}$.

If we make sure not to change $\mathcal{A}$'s view of the game in the case $k = g^{x \cdot y}$ in this process, we can achieve an advantage of about $\Pr[F_{\mathrm{DH}}]/N^2$, where one factor $1/N$ arises from guessing a and another from guessing u_j. Unfortunately, this would yield no improvement over the result from [1].

To avoid this issue, we can use random self-reducability of the DDH problem and avoid guessing u_j. Instead of embedding g^y into a single encryption edge, we embed it into all d encryption edges. To get a uniformly random exponent from y we set $y_j = y + r_j \mod q$ with $r_j \leftarrow [q]$. Given $g^{x \cdot y_j}$, we can easily compute $g^{x \cdot y}$:

$$g^{x \cdot y_j} = g^{x \cdot (y + r_j)} = g^{x \cdot y} \cdot g^{x \cdot r_j} \iff g^{x \cdot y} = g^{x \cdot y_j} \cdot \underbrace{((g^x)^{r_j})^{-1}}_{=: R_j}.$$

Now to determine whether k is the real Diffie-Hellman key, we check whether $q_i \cdot R_j = k$ for some i, j. This yields an advantage of about $\Pr[F_{\mathrm{DH}}]/N$ (and a slightly larger runtime). The details of the proof can be found in Section B.4 of the appendix.

References

1. Alwen, J., Capretto, M., Cueto, M., Kamath, C., Klein, K., Markov, I., Pascual-Perez, G., Pietrzak, K., Walter, M., Yeo, M.: Keep the dirt: Tainted TreeKEM, adaptively and actively secure Continuous Group Key Agreement. Cryptology ePrint Archive, Paper 2019/1489 (2019), https://eprint.iacr.org/2019/1489
2. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Security analysis and improvements for the IETF MLS standard for group messaging. Cryptology ePrint Archive, Paper 2019/1189 (2019), https://eprint.iacr.org/2019/1189
3. Alwen, J., Coretti, S., Dodis, Y., Tselekounis, Y.: Modular design of secure group messaging protocols and the security of MLS. Cryptology ePrint Archive, Paper 2021/1083 (2021), https://eprint.iacr.org/2021/1083
4. Alwen, J., Jost, D., Mularczyk, M.: On the insider security of MLS. Cryptology ePrint Archive, Paper 2020/1327 (2020), https://eprint.iacr.org/2020/1327
5. Balbás, D., Collins, D., Gajland, P.: WhatsUpp with Sender Keys? Analysis, improvements and security proofs. Cryptology ePrint Archive, Paper 2023/1385 (2023), https://eprint.iacr.org/2023/1385
6. Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., Cohn-Gordon, K.: The Messaging Layer Security (MLS) Protocol. RFC 9420 (Jul 2023). https://doi.org/10.17487/RFC9420, https://www.rfc-editor.org/info/rfc9420
7. Bellare, M., Rogaway, P.: Random oracles are practical: a paradigm for designing efficient protocols. In: Proceedings of the 1st ACM Conference on Computer and Communications Security. p. 62–73. CCS '93, Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/168588.168596
8. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Public Key Cryptography-PKC 2006: 9th International Conference on Theory and Practice in Public-Key Cryptography, New York, NY, USA, April 24-26, 2006. Proceedings 9. pp. 207–228. Springer (2006)
9. Bhargavan, K., Barnes, R., Rescorla, E.: TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). Research report, Inria Paris (May 2018), https://inria.hal.science/hal-02425247

10. Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A formal security analysis of the signal messaging protocol. In: 2017 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 451–466 (2017). https://doi.org/10.1109/EuroSP.2017.27
11. Hogben, G.: An important step towards secure and interoperable messaging. https://security.googleblog.com/2023/07/an-important-step-towards-secure-and.html (2023), accessed: 2023-11-01
12. IETF: Support for MLS. https://www.ietf.org/blog/support-for-mls-2023/ (2023), accessed: 2023-11-01
13. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, Third Edition. Chapman & Hall/CRC, 3rd edn. (2020)
14. Panjwani, S.: Tackling adaptive corruptions in multicast encryption protocols. In: Proceedings of the 4th Conference on Theory of Cryptography. p. 21–40. TCC'07, Springer-Verlag, Berlin, Heidelberg (2007)
15. Perrin, T., Marlinspike, M.: The Double Ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/ (2016), accessed: 2024-03-05

A Basic Cryptographic Definitions

A.1 Encryption schemes

Private-key encryption

Definition 10 (Private-key encryption [13, Definition 3.7]). *Let κ denote the security parameter. A* private-key encryption scheme *Π consists of three probabilistic polynomial-time algorithms* $(\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$ *such that:*

1. *The* key-generation algorithm Gen *takes as input 1^κ (in unary) and outputs a key k. We will write $k \leftarrow \mathrm{Gen}(1^\kappa)$.*
2. *The* encryption algorithm Enc *takes as input a key k and a message $m \in \{0,1\}^*$, or $m \in \{0,1\}^{\leq l(\kappa)}$ for some function l if the message space is finite, and outputs a ciphertext c. We write this as $c \leftarrow \mathrm{Enc}_k(m)$.*
3. *The deterministic* decryption algorithm Dec *takes as input a key k and a ciphertext c, and outputs a message m or $\perp$ (denoting an error). We write this as $m = \mathrm{Dec}_k(c)$.*

It is required that for every κ, every key k output by Gen*, and every message m, it holds that $\Pr[\mathrm{Dec}_k(\mathrm{Enc}_k(m)) = m] = 1$ (where the probability is over the randomness of* Enc_k*).*

Public-key encryption In the following definition we will be more explicit about the randomness used by the algorithm Gen, as we will require a way to provide the randomness as input.

Definition 11 (Public-key encryption [13, Definition 12.1]). *Let η denote the security parameter. A* public-key encryption scheme *Π consists of three probabilistic polynomial-time algorithms* $(\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$ *such that:*

1. *The* key-generation algorithm Gen *takes as input* 1^η *(in unary) and outputs a pair of keys* (pk, sk) *(a public and private key). We will write* $(pk, sk) \leftarrow \mathrm{Gen}(1^\eta)$*.*
   The public key defines a message space $\mathcal{M}_{pk}$*.*
   The algorithm samples $\rho(\eta)$ *uniformly random bits to make randomized decisions for some function* ρ *polynomial in* η*. The sequence of random bits* $r \in \{0,1\}^{\rho(\eta)}$ *to be used by the algorithm may also be provided as input. We write this as* $(pk, sk) = \mathrm{Gen}(1^\eta, r)$ *to emphasize the fact that the output is deterministic. The distribution over key pairs output by sampling* $r \leftarrow \{0,1\}^{\rho(\eta)}$ *and running* $\mathrm{Gen}(1^\eta, r)$ *is identical to the distribution over key pairs output by running* $\mathrm{Gen}(1^\eta)$*.*
2. *The* encryption algorithm Enc *takes as input a public key pk and a message* $m \in \mathcal{M}_{pk}$*, and outputs a ciphertext c. We write this as* $c \leftarrow \mathrm{Enc}_{pk}(m)$*.*
3. *The deterministic* decryption algorithm Dec *takes as input a private key sk and a ciphertext c, and outputs a message m or* $\perp$ *(denoting an error). We write this as* $m = \mathrm{Dec}_{sk}(c)$*.*

It is required that for every η*, every key* (pk, sk) *output by* Gen*, and every message m, it holds that* $\Pr[\mathrm{Dec}_{sk}(\mathrm{Enc}_{pk}(m)) = m] = 1$ *(where the probability is over the randomness of* Enc_{pk}*).*

A.2 Security definitions

Definition 12 (The IND-CPA game). *Let* κ *denote the security parameter and let* Π *a private-key encryption scheme. Define the game* $\mathrm{Game}^{\mathrm{IND-CPA}}_{\Pi,\kappa}(\mathcal{A})$ *for an adversary* $\mathcal{A}$*:*

1. *A key* $k \leftarrow \mathrm{Gen}(1^\kappa)$ *is generated.*
2. *The adversary* $\mathcal{A}$ *is given oracle access to* $\Pi.\mathrm{Enc}_k$*, and outputs a pair of messages* m_0, m_1 *of the same length.*
3. *A bit* $b \leftarrow \{0,1\}$ *is sampled and* $\mathcal{A}$ *is given a ciphertext* $c \leftarrow \mathrm{Enc}_k(m_b)$*. (*$\mathcal{A}$ *continues to have oracle access to* $\Pi.\mathrm{Enc}_k$*.)*
4. $\mathcal{A}$ *outputs a bit* b'*. The output of the game is defined to be* 1 *if* $b' = b$*, and* 0 *otherwise.*

Definition 13 (IND-CPA security [13, Definition 3.21]). *For functions* t, ε *in the security parameter* κ*, a private-key encryption scheme* Π *is* (t, ε)*-*IND-CPA-secure *if for all* κ*, for any adversary* $\mathcal{A}$ *running in time* $t(\kappa)$ *we have*

$$\mathrm{Adv}^{\mathrm{IND-CPA}}_{\Pi,\kappa}(\mathcal{A}) \coloneqq 2 \cdot \left(\Pr\left[\mathrm{Game}^{\mathrm{IND-CPA}}_{\Pi,\kappa}(\mathcal{A}) = 1\right] - \frac{1}{2}\right) \leq \varepsilon(\kappa).$$

We will make use of a weaker form of security called “indistinguishability in the presence of an eavesdropper” [13] and will refer to it as “EAV security”. It is identical to IND-CPA security with the sole exception that the adversary does not have access to an encryption oracle.

Definition 14 (The EAV game). *Let κ denote the security parameter and let Π a private-key encryption scheme. Define the game* $\mathrm{Game}^{\mathrm{EAV}}_{\Pi,\kappa}(\mathcal{A})$ *for an adversary $\mathcal{A}$:*

1. *A key $k \leftarrow \mathrm{Gen}(1^\kappa)$ is generated.*
2. *The adversary $\mathcal{A}$ outputs a pair of messages m_0, m_1 of the same length.*
3. *A bit $b \leftarrow \{0,1\}$ is sampled and $\mathcal{A}$ is given a ciphertext $c \leftarrow \mathrm{Enc}_k(m_b)$.*
4. *$\mathcal{A}$ outputs a bit b'. The output of the game is defined to be 1 if $b' = b$, and 0 otherwise.*

Definition 15 (EAV security [13, Definition 3.8]). *A private-key encryption scheme Π is* (t,ε)-EAV-secure *if for all κ, for any adversary $\mathcal{A}$ running in time $t(\kappa)$ we have*

$$\mathrm{Adv}^{\mathrm{EAV}}_{\Pi,\kappa}(\mathcal{A}) \coloneqq 2 \cdot \left(\Pr\big[\mathrm{Game}^{\mathrm{EAV}}_{\Pi,\kappa}(\mathcal{A}) = 1\big] - \frac{1}{2}\right) \leq \varepsilon(\kappa).$$

Lemma 4. *Let Π a private-key encryption scheme. If Π is (t,ε)-IND-CPA-secure, then Π is (t,ε)-EAV-secure.*

Proof. This follows immediately from the fact that any EAV adversary is also an IND-CPA adversary.

When analyzing the advantage of an adversary we may make use of the following well known equality.

Lemma 5. *Let X a Bernoulli random variable and $b \leftarrow \{0,1\}$ (where X and b are not necessarily independent). Then for $x \in \{0,1\}$*

$$2 \cdot \left(\Pr[X = b] - \frac{1}{2}\right) = \Pr[X = x \mid b = x] - \Pr[X = x \mid b = 1 - x].$$

In particular, if $\mathcal{A}$ is an adversary with output in $\{0,1\}$ playing a game where a bit $b \leftarrow \{0,1\}$ is sampled, then for $x \in \{0,1\}$

$$2 \cdot \left(\Pr[b \leftarrow \mathcal{A}] - \frac{1}{2}\right) = \Pr[x \leftarrow \mathcal{A} \mid b = x] - \Pr[x \leftarrow \mathcal{A} \mid b = 1 - x]. \tag{4}$$

Proof. Let $x \in \{0,1\}$. We have

$$\begin{aligned} 2 \cdot \left(\Pr[X = b] - \frac{1}{2}\right) &= 2 \cdot \left(\Pr[X = x \mid b = x] \cdot \frac{1}{2} + \Pr[X = 1 - x \mid b = 1 - x] \cdot \frac{1}{2} - \frac{1}{2}\right) \\ &= \Pr[X = x \mid b = x] + \Pr[X = 1 - x \mid b = 1 - x] - 1 \\ &= \Pr[X = x \mid b = x] - (1 - \Pr[X = 1 - x \mid b = 1 - x]) \\ &= \Pr[X = x \mid b = x] - \Pr[X = x \mid b = 1 - x]. \end{aligned}$$

A.3 The Random Oracle Model

We will work in the commonly used Random Oracle Model (ROM) to prove our results. We refer the reader to [13, Chapter 6.5] for an informal overview of the ROM and to [7] for the original work that introduced the model. The ROM introduces the concept of a *random oracle*. If a function $H : A \to B$ is modelled as a random oracle, then certain assumptions are made about what an adversary $\mathcal{A}$ knows about H and how it interacts with it:

- From $\mathcal{A}$'s perspective, H is a black-box function. The only way for $\mathcal{A}$ to interact with H is for it to provide a value $a \in A$ and get back $H(a)$, and this is the only way for $\mathcal{A}$ to learn $H(a)$. We say that $\mathcal{A}$ *queries* $H(a)$ or that $\mathcal{A}$ *queries* H *for* a. This well-defined interface of $\mathcal{A}$ to H implies that a reduction can extract the queries that $\mathcal{A}$ makes to H.
- From $\mathcal{A}$'s perspective, H is a random variable, sampled u.a.r. from the set of all functions from A to B. Thus, if $\mathcal{A}$ queries H for some $a \in A$ that it has not queried before, the value $H(a)$ is a random variable uniformly distributed in B from $\mathcal{A}$'s perspective.

We do not rely on the property known as "programmability" in this work.

B Tighter GSD security

B.1 Proof of Lemma 1

Proof (of Lemma 1). Note that since the message space is finite, the time to encrypt a message is bounded. As outlined before Lemma 1, the lemma follows from a simple hybrid argument. Let q a function in κ, let κ arbitrary and let $\mathcal{A}$ an arbitrary MIS-EAV adversary running in time $\tilde{t}(\kappa)$ and making at most $q(\kappa)$ queries. Define the sequence of hybrid games $H_0, \ldots, H_q$ where in the game H_i the first i encryptions given to the adversary encrypt m_1 and all remaining encryptions encrypt m_0. We will write

$$\Pr[0 \leftarrow \mathcal{A} \mid H_i]$$

for the probability of $\mathcal{A}$ outputting 0 when playing the hybrid game H_i.

Let $i \in [q]$. Construct an EAV adversary $\mathcal{A}'$ that behaves as follows:

1. $\mathcal{A}'$ runs $\mathcal{A}$ and gets q, m_0, m_1.
2. $\mathcal{A}'$ outputs the messages m_0, m_1 and gets a ciphertext c from the challenger.
3. $\mathcal{A}'$ gives the ciphertexts $c_1, \ldots, c_q$ to $\mathcal{A}$ where

$$c_j = \begin{cases} \mathit{\Pi}.\mathrm{Enc}_{k_j}(m_1) & i < j \\ c & i = j \\ \mathit{\Pi}.\mathrm{Enc}_{k_j}(m_0) & i > j \end{cases}$$

and $k_j \leftarrow \mathit{\Pi}.\mathrm{Gen}(1^\kappa)\ \forall j$.

4. $\mathcal{A}'$ outputs whatever bit $\mathcal{A}$ outputs.

Now consider the value of the bit b sampled in $\mathrm{Game}^{\mathrm{EAV}}_{\Pi,\kappa}(\mathcal{A}')$. If $b = 0$, then the first $i-1$ ciphertexts that $\mathcal{A}$ received were encryptions of m_1 and the remaining ciphertexts were encryptions of m_0, where all encryptions were under keys sampled independently with Π.Gen. Thus, from the view of $\mathcal{A}$ everything followed the same distribution as in the game H_{i-1} and

$$\Pr[0 \leftarrow \mathcal{A}' \mid b = 0] = \Pr[0 \leftarrow \mathcal{A} \mid H_{i-1}].$$

Analogously, in the case $b = 1$ the first i ciphertexts received by $\mathcal{A}$ were encryptions of m_1 and the rest encryptions of m_0, so

$$\Pr[0 \leftarrow \mathcal{A}' \mid b = 1] = \Pr[0 \leftarrow \mathcal{A} \mid H_i].$$

Then

$$\begin{aligned}\Pr[0 \leftarrow \mathcal{A} \mid H_{i-1}] - &\Pr[0 \leftarrow \mathcal{A} \mid H_i] \\ &= \Pr[0 \leftarrow \mathcal{A}' \mid b = 0] - \Pr[0 \leftarrow \mathcal{A}' \mid b = 1] \\ &\overset{(4)}{=} \mathrm{Adv}^{\mathrm{EAV}}_{\Pi,\kappa}(\mathcal{A}') \\ &\leq \varepsilon\end{aligned} \tag{5}$$

by (t, ε)-EAV security of Π since $\mathcal{A}'$ runs in time $\tilde{t} + \mathcal{O}(q \cdot (t_{\mathrm{Gen}} + t_{\mathrm{Enc}})) = t$.

Now let b be the bit sampled in the MIS-EAV game. Notice that

$$\Pr[0 \leftarrow \mathcal{A} \mid b = 0] = \Pr[0 \leftarrow \mathcal{A} \mid H_0]$$

and

$$\Pr[0 \leftarrow \mathcal{A} \mid b = 1] = \Pr[0 \leftarrow \mathcal{A} \mid H_q].$$

Then

$$\begin{aligned}\mathrm{Adv}^{\mathrm{MIS-EAV}}_{\Pi,\kappa}(\mathcal{A}) &\overset{(4)}{=} \Pr[0 \leftarrow \mathcal{A} \mid b = 0] - \Pr[0 \leftarrow \mathcal{A} \mid b = 1] \\ &= \Pr[0 \leftarrow \mathcal{A} \mid H_0] - \Pr[0 \leftarrow \mathcal{A} \mid H_q] \\ &= \sum_{i=1}^{q} \Pr[0 \leftarrow \mathcal{A} \mid H_{i-1}] - \Pr[0 \leftarrow \mathcal{A} \mid H_i] \\ &\overset{(5)}{\leq} q \cdot \varepsilon.\end{aligned}$$

B.2 Proof of Lemma 2

Proof (of Lemma 2). As already motivated after Lemma 2, we construct an MIS-EAV adversary $\mathcal{A}'$ to derive the bound. $\mathcal{A}'$ behaves as follows:

1. $\mathcal{A}'$ runs $\mathcal{A}$ to get n and D and initializes the GSD graph, seeds and the set of edges and corrupted nodes as in step 1. of the SD-GSD game.

2. $\mathcal{A}'$ samples $w \leftarrow [n], s \leftarrow \{0,1\}^\rho$ and gives δ and the messages s_w, s to the challenger. Let $c_1, \dots, c_\delta$ the encryptions it gets back.
3. $\mathcal{A}'$ faithfully simulates the SD-GSD game to $\mathcal{A}$ with the following exception: Whenever $\mathcal{A}$ makes a query of the form $\text{encrypt}(u, w)$ for any u, $\mathcal{A}'$ replies with $\langle g^x, c_i \rangle$ where $x \leftarrow [q]$ and i is the index of the next ciphertext (from step 2.) not yet used.
   All random oracles queries are simulated by sampling the output of the oracle u.a.r. for new queries and using the value first sampled for repeated queries.
   During the simulation $\mathcal{A}'$ also pays attention to the following:
   - If any of the following events occur, $\mathcal{A}'$ aborts the simulation and outputs 1:
     - $\mathcal{A}$ queries H_{DH} for a hidden Diffie-Hellman key
     - $\mathcal{A}$ queries H_{gen} or H_{dep} for a hidden seed that is not s_w
     - $\mathcal{A}$ queries $\text{corrupt}(u)$ for some node u such that w is no longer safe
   - If $\mathcal{A}$ queries $H_{\text{gen}}(s_w)$ or $H_{\text{dep}}(s_w)$, $\mathcal{A}'$ aborts the simulation and outputs 0. This is the only point at which $\mathcal{A}'$ outputs 0.

   If the simulation arrives to the point where $\mathcal{A}$ outputs its guess (step 4. of the SD-GSD game), then $\mathcal{A}'$ outputs 1.

The advantage of $\mathcal{A}'$ is given by

$$\text{Adv}_{\Pi_s,\eta}^{\text{MIS-EAV}}(\mathcal{A}') \overset{(4)}{=} \Pr[0 \leftarrow \mathcal{A}' \mid b = 0] - \Pr[0 \leftarrow \mathcal{A}' \mid b = 1], \tag{6}$$

where b is the bit sampled by the MIS-EAV challenger.

First, we will show that

$$\Pr[0 \leftarrow \mathcal{A}' \mid b = 0] \geq \frac{\Pr\left[Q_{\text{s}} \wedge \overline{F_{\text{DH}}}\,\right]}{N}. \tag{7}$$

Let $E = Q_{\text{s}} \wedge \overline{F_{\text{DH}}}$ and let E' the same event in the SD-GSD game simulated to $\mathcal{A}$ during an execution of $\text{Game}_{\Pi_s,\eta}^{\text{MIS-EAV}}(\mathcal{A}')$ with $b = 0$. In the following while showing (7) we will implicitly assume that $b = 0$ when referring to the game simulated to $\mathcal{A}$ by $\mathcal{A}'$. On a high level (7) holds due to the fact that as long as the game has not been aborted the encryptions $\mathcal{A}$ receives from $\mathcal{A}'$ are indistinguishable from what it would get in the real SD-GSD game and we get a factor $\frac{1}{N}$ from guessing the node that triggered E. However, showing this requires a few steps.

Consider a modification of the SD-GSD game G_1 where the game is aborted whenever one of the following events occurs, where for all these events $\mathcal{A}'$ would also abort the simulation:

- $\mathcal{A}$ queries H_{DH} for a hidden Diffie-Hellman key
- $\mathcal{A}$ queries H_{gen} or H_{dep} for a hidden seed

(Since we are not interested in the output of the game we can define *aborting the game* as the game ending with output 0.) The game G_1 is something between the real SD-GSD game and what $\mathcal{A}'$ simulates to $\mathcal{A}$. The only difference in when G_1

aborts compared to the game simulated by $\mathcal{A}'$ is that we aren't paying attention to some specific node w remaining safe. Aborting the game in this way does not alter the probability of $\mathcal{A}$ triggering the event E in G_1, since in either case when the game is aborted either E or $\overline{E}$ is already known to hold:

- If $\mathcal{A}$ queries H_{DH} for a hidden Diffie-Hellman key, then it triggers Q_{DH} and Q_{s} has not been triggered before since this would have caused the game to be aborted. Thus, $\mathcal{A}$ triggered F_{DH} and $Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}$ cannot hold in this execution of the game.
- If $\mathcal{A}$ queries H_{gen} or H_{dep} for a hidden seed, then this triggers Q_{s}. Moreover, $\overline{F_{\mathrm{DH}}}$ also holds at this moment since the game would have aborted earlier if Q_{DH} had already been triggered. Thus, $Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}$ holds.

Let E_1 the same event as E in the game G_1. As argued above we have

$$\Pr[E_1] = \Pr[E]. \tag{8}$$

Now consider a game G_2 which is a modification of the game G_1 where at the beginning of the game $w_2 \leftarrow [n]$ is sampled and the game also aborts if $\mathcal{A}$ queries $\mathrm{corrupt}(u)$ for some node u such that w_2 is no longer safe, just as in the game simulated by $\mathcal{A}'$. The game G_2 is again something between the game G_1 and what $\mathcal{A}'$ simulates to $\mathcal{A}$. We also modify G_1 such that it also samples $w_1 \leftarrow [n]$ at the beginning of the game. This does not change the fact that (8) holds as the sampling of w_1 has no effect on the execution of the game.

Let E_2 and E' the events corresponding to E in the game G_2 and the game simulated by $\mathcal{A}'$, respectively. We further introduce a new random variable W to analyze each game where

$$W = \begin{cases} 0 & \overline{E} \\ x & E \text{ was triggered at node } x \end{cases}$$

(if x is not unique we choose the node with lowest identifier). Let W_1, W_2 and W' be the corresponding random variables in game G_1, game G_2 and the game simulated by $\mathcal{A}'$. Consider the probability $\Pr[W_1 = w_1 \mid E_1]$. The node w_1 is sampled independently and does not affect the execution of the game. Therefore, in an execution where E_1 occurs and the GSD graph has size n (so $W_1 \in [n]$), we correctly guess $W_1 = w_1$ with probability exactly $\frac{1}{n} \geq \frac{1}{N}$. Thus

$$\Pr[W_1 = w_1 \mid E_1] \geq \frac{1}{N}$$

and combining this with (8) we get

$$\begin{aligned} \Pr[W_1 = w_1] &= \Pr[W_1 = w_1 \wedge E_1] \\ &= \Pr[W_1 = w_1 \mid E_1] \cdot \Pr[E_1] \\ &\geq \frac{1}{N} \cdot \Pr[E]. \end{aligned} \tag{9}$$

Analogously to the argument used to justify (8), we can argue that

$$\Pr[W_1 = w_1] = \Pr[W_2 = w_2]. \tag{10}$$

The only difference from G_1 to G_2 is that G_2 aborts when w_2 is no longer safe. But if w_2 is no longer safe then we know that $W_2 \neq w_2$ (if $W_2 = w_2$ the game would have already aborted when w_2's seed was queried while it was safe). Thus, (10) indeed holds.

We now show an analogous result comparing the game G_2 to the game simulated by $\mathcal{A}'$:

$$\Pr[W_2 = w_2] = \Pr[W' = w]. \tag{11}$$

Consider how G_2 differs from the game simulated by $\mathcal{A}'$. Both games abort at exactly the same events (this is easy to see). They only differ in how $\mathcal{A}'$ answers queries $\mathrm{encrypt}(u, w)$ for any u. In G_2 such a query is answered with a ciphertext $\langle g^x, c\rangle$ where $x \leftarrow [q], c \leftarrow \mathit{\Pi}_s.\mathrm{Enc}_k(s_w)$ and $k = H_{\mathrm{DH}}(pk_u^x)$. $\mathcal{A}'$ answers such a query with $\langle g^{x'}, c'\rangle$ where $x' \leftarrow [q], c' \leftarrow \mathit{\Pi}_s.\mathrm{Enc}_{k'}(s_w)$ and $k' \leftarrow \{0,1\}^\eta$. Now notice that as long as the game G_2 is ongoing, pk_u^x is a hidden Diffie-Hellman key and $\mathcal{A}$ has not queried pk_u^x to H_{DH}. If it had, then the game would have already aborted. Therefore, from $\mathcal{A}$'s view k follows the same distribution as k'. Thus, overall the game G_2 and the game simulated by $\mathcal{A}'$ are indistinguishable to $\mathcal{A}$ and (11) holds.

Finally, notice that if the event $W' = w$ occurred, then $\mathcal{A}'$ outputs 0. Then we have

$$\begin{aligned}
\Pr[0 \leftarrow \mathcal{A}' \mid b = 0] &\geq \Pr[W' = w] \\
&\overset{(11)}{=} \Pr[W_2 = w_2] \\
&\overset{(10)}{=} \Pr[W_1 = w_1] \\
&\overset{(9)}{\geq} \frac{\Pr[E]}{N} \\
&= \frac{\Pr\left[Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}\,\right]}{N},
\end{aligned}$$

as promised.

Second, returning to (6), we can more easily show that $\Pr[0 \leftarrow \mathcal{A}' \mid b = 1]$ is negligible. In the SD-GSD game simulated to $\mathcal{A}$ during an execution of $\mathrm{Game}_{\mathit{\Pi}_s,\eta}^{\mathrm{MIS-EAV}}(\mathcal{A}')$ with $b = 1$, the seed s_w is a random variable independent of any information given to $\mathcal{A}$:

- the game aborts when w becomes unsafe, so s_w cannot be learned by querying $\mathrm{corrupt}(w)$ or by querying $H_{\mathrm{dep}}(s_p)$ for an unsafe node p where (p, w) is a seed dependency
- querying $H_{\mathrm{dep}}(s_p)$ for a safe node p where (p, w) is a seed dependency results in the game being aborted and by virtue of H_{dep} being a random oracle, from $\mathcal{A}$'s perspective s_w follows the same distribution regardless of whether there is a seed dependency (p, w) or not

- with $b = 1$ queries encrypt(u, w) yield encryptions of s instead of s_w

Therefore, for any seed s' that $\mathcal{A}$ queries to H_{gen} or H_{dep} we have

$$\Pr[s_w = s'] = \frac{1}{2^\rho}.$$

Thus, by a union bound we have

$$\Pr[0 \leftarrow \mathcal{A}' \mid b = 1] \leq \frac{m_{\text{s}}}{2^\rho}. \tag{12}$$

Combining (6), (7) and (12) we get

$$\begin{aligned} \mathrm{Adv}^{\text{MIS}-\text{EAV}}_{\mathit{\Pi}_s,\eta}(\mathcal{A}') &= \Pr[0 \leftarrow \mathcal{A}' \mid b = 0] - \Pr[0 \leftarrow \mathcal{A}' \mid b = 1] \\ &\geq \frac{\Pr\big[Q_{\text{s}} \wedge \overline{F_{\text{DH}}}\,\big]}{N} - \frac{m_{\text{s}}}{2^\rho}. \end{aligned} \tag{13}$$

Furthermore, going through the details yields that $\mathcal{A}'$ runs in time

$$\begin{aligned} t_{\mathcal{A}'} \coloneqq \tilde{t} + \mathcal{O}\big(\rho \cdot t_{\text{sample}} \cdot m_{\text{s}} + (\gamma + \eta \cdot t_{\text{sample}}) \cdot m_{\text{DH}} \\ + N \cdot (\rho \cdot t_{\text{sample}} + t_{\mathit{\Pi}_{\text{DH}}.\text{Gen}}) \\ + N^2 \cdot t_{\mathit{\Pi}_{\text{DH}}.\text{Enc}}\big) \end{aligned}$$

(note that $t_{\mathcal{A}'}$ is a constant). Using that $\delta \leq N, t_{\mathit{\Pi}_s.\text{Gen}} \leq \mathcal{O}(\eta \cdot t_{\text{sample}})$, $t_{\mathit{\Pi}_s.\text{Enc}} \leq t_{\mathit{\Pi}_{\text{DH}}.\text{Enc}}$ (as encrypting with $\mathit{\Pi}_{\text{DH}}$ involves an encryption with $\mathit{\Pi}_s$), the definition of $\tilde{t}$, with appropriately chosen constants we have

$$t_{\mathcal{A}'} \leq t - \mathcal{O}(\delta \cdot (t_{\mathit{\Pi}_s.\text{Gen}} + t_{\mathit{\Pi}_s.\text{Enc}})).$$

By Lemma 1 $\mathit{\Pi}_s$ is $(t - \mathcal{O}(\delta \cdot (t_{\mathit{\Pi}_s.\text{Gen}} + t_{\mathit{\Pi}_s.\text{Enc}})), \delta \cdot \varepsilon_{\text{EAV}}, \delta)$-MIS-EAV-secure, so

$$\mathrm{Adv}^{\text{MIS}-\text{EAV}}_{\mathit{\Pi}_s,\eta}(\mathcal{A}') = \delta \cdot \varepsilon_{\text{EAV}}. \tag{14}$$

Finally, if we now combine (13) and (14) we get

$$\begin{gathered} \frac{\Pr\big[Q_{\text{s}} \wedge \overline{F_{\text{DH}}}\,\big]}{N} - \frac{m_{\text{s}}}{2^\rho} \leq \delta \cdot \varepsilon_{\text{EAV}} \\ \Longleftrightarrow \\ \Pr\big[Q_{\text{s}} \wedge \overline{F_{\text{DH}}}\,\big] \leq \delta \cdot N \cdot \varepsilon_{\text{EAV}} + \frac{m_{\text{s}} \cdot N}{2^\rho}, \end{gathered}$$

as was to prove.

B.3 Tighter MIS-EAV security for certain schemes

In our reduction from MIS-EAV security to EAV security (Lemma 1) we applied a general hybrid argument. It is also tempting to try a more direct approach. The EAV and MIS-EAV games seem less far apart than IND-CPA for single and multiple encryptions: All additional encryptions in the MIS-EAV game encrypt the same message, with the only difference being that each encryption is performed using a fresh key. If only we could take a single encryption $c \leftarrow \mathrm{Enc}_k(m)$ and from it produce several encryptions $c_i \leftarrow \mathrm{Enc}_{k_i}(m)$ for $k_i \leftarrow \mathrm{Gen}(1^\kappa)$ (without knowing k or m), then the additional encryptions would leak no new information to the adversary, and we would have a tight bound on MIS-EAV security from EAV security. There is a simple EAV secure scheme that achieves the above property: the one-time pad. Given an encryption $c = k \oplus m$, we can just sample $k' \leftarrow \{0,1\}^\kappa$ and compute the ciphertext $c' = c \oplus k' = (k \oplus k') \oplus m$, an encryption of m under the uniformly random key $k \oplus k'$. In the following, we formalize this property of a private-key encryption scheme and use it to prove the desired bound on MIS-EAV security.

Definition 16 (Key-rerandomizability). *Let κ denote the security parameter and let $\Pi = (\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$ a private-key encryption scheme. Π is* key-rerandomizable *if there exists a probabilistic polynomial-time algorithm* ReRan *achieving the following: Let $\kappa, k \leftarrow \mathrm{Gen}(1^\kappa)$, m in the message space and $c \leftarrow \mathrm{Enc}_k(m)$ be arbitrary but fixed*[6]*. Then the distribution over ciphertexts as defined by computing $c' \leftarrow \mathrm{ReRan}(1^\kappa, c)$ is identical to the distribution over ciphertexts resulting from the process of first sampling $k' \leftarrow \mathrm{Gen}(1^\kappa)$ and then computing a ciphertext $c' \leftarrow \mathrm{Enc}_{k'}(m)$.*

Example As outlined above, the one-time pad is an example of a key-rerandomizable encryption scheme.

Q: Is there a key-rerandomizable IND-CPA secure scheme? If yes, this would imply a key-rerandomizable AE scheme using the encrypt-then-authenticate paradigm, since a rerandomized tag can easily produced for the ciphertext by sampling a fresh MAC key.

The key idea underlying the proof of the following Lemma was already provided at the beginning of this section.

Lemma 6. *Let Π a key-rerandomizable private-key encryption scheme with finite message space. Let* ReRan *the corresponding algorithm to rerandomize ciphertexts and t_{ReRan} an upper bound for the runtime of* ReRan*. If Π is (t, ε)-EAV-secure, then for all $q \in \mathbb{N}$, Π is $(\tilde{t}, \varepsilon, q)$-MIS-EAV-secure with $\tilde{t} = t - \mathcal{O}(q \cdot t_{\mathrm{ReRan}})$.*

Proof. Note that since the message space and thus the ciphertext space is finite, the runtime of ReRan is indeed bounded. Let κ arbitrary. Let $\mathcal{A}$ an MIS-EAV adversary running in time $\tilde{t}(\kappa)$ and making at most $q(\kappa)$ queries. We construct an EAV adversary $\mathcal{A}'$ that behaves as follows:

1. $\mathcal{A}'$ runs $\mathcal{A}$ to get the number of queries q and messages m_0, m_1.
2. $\mathcal{A}'$ gives m_0, m_1 to the challenger and receives the ciphertext c_1.
3. $\mathcal{A}'$ computes ciphertexts $c_2 \leftarrow \mathrm{ReRan}(1^\kappa, c_1), \ldots, c_q \leftarrow \mathrm{ReRan}(1^\kappa, c_1)$ (with independent runs of ReRan).

[6] Here we are quantifying over all possible keys k and ciphertexts c that can be output by $\mathrm{Gen}(1^\kappa)$ and $\mathrm{Enc}_k(m)$.

4. $\mathcal{A}'$ gives the ciphertexts $c_1, \ldots, c_q$ to $\mathcal{A}$.
5. $\mathcal{A}'$ outputs whatever bit $\mathcal{A}$ outputs.

We apply the properties of ReRan given in Definition 16 to show that the game simulated to $\mathcal{A}$ is distributed identically to the MIS-EAV game. For this we need only show that the ciphertexts $c_1, \ldots, c_q$ given to $\mathcal{A}$ in the simulation are distributed identically to the ciphertexts $c'_1, \ldots, c'_q$ that $\mathcal{A}$ would get in the real MIS-EAV game. It is immediate that c_1 is distributed identically to c'_1. Now let $i \in \{2, \ldots, q\}$. By Definition 16 ReRan(c) outputs a ciphertext encrypting m_b (where b is the bit chosen by the EAV challenger) distributed identically to a ciphertext encrypting m_b output by the MIS-EAV challenger. Thus, indeed for any i, c_i is distributed identically to c'_i and the claim holds. Therefore

$$\mathrm{Adv}^{\mathrm{MIS-EAV}}_{\Pi,\kappa}(\mathcal{A}) = \mathrm{Adv}^{\mathrm{EAV}}_{\Pi,\kappa}(\mathcal{A}'). \tag{15}$$

Because $\mathcal{A}'$ is an EAV adversary running in time $\tilde{t} + \mathcal{O}(q \cdot t_{\mathrm{ReRan}}) = t$ we know that

$$\mathrm{Adv}^{\mathrm{EAV}}_{\Pi,\kappa}(\mathcal{A}') \leq \varepsilon(\kappa),$$

which together with (15) concludes the proof.

By assuming a key-rerandomizable encryption scheme and applying Lemma 6 on the facing page instead of the hybrid argument (Lemma 1) in the proof of Lemma 2, we can drop the δ factor in the bound. This also allows us to drop the δ factor in Theorem 2 on page 16.

Corollary 1. *Recall the setting of Theorem 2. If the private-key encryption scheme Π_s is additionally key-rerandomizable, then the bound in Lemma 2 can be improved to*

$$\Pr\left[Q_{\mathrm{s}} \wedge \overline{F_{\mathrm{DH}}}\,\right] \leq N \cdot \varepsilon_{\mathrm{EAV}} + \frac{m_{\mathrm{s}} \cdot N}{2^{\rho}}$$

and the bound $\tilde{\varepsilon}$ on the success probability of an SD-GSD adversary thus improved to

$$\tilde{\varepsilon} = 2 \cdot N \cdot (\varepsilon_{\mathrm{EAV}} + \varepsilon_{\mathrm{DDH}}) + \frac{2 \cdot m_{\mathrm{DH}} \cdot N^2}{q} + \frac{m_{\mathrm{s}} \cdot N}{2^{\rho-1}}$$

(with appropriate changes to the runtime $\tilde{t}$).

B.4 Proof of Lemma 3

Proof (of Lemma 3). As outlined after Lemma 3 we use $\mathcal{A}$ to construct a DDH adversary $\mathcal{A}'$:

1. $\mathcal{A}'$ gets h_1, h_2 and k from the DDH challenger.
2. $\mathcal{A}'$ runs $\mathcal{A}$ to get n and D, samples $a \leftarrow [n]$ and initializes the GSD graph, seeds and the set of edges and corrupted nodes as in step 1. of the SD-GSD game, with the sole exception that $pk_a = h_1$ (as opposed to setting it to the public key output by $\Pi_{\mathrm{DH}}.\mathrm{Gen}(1^{\eta}, H_{\mathrm{gen}}(s_a))$).

3. $\mathcal{A}'$ faithfully simulates the SD-GSD game to $\mathcal{A}$ with the following exception: For the j-th query encrypt(a, u_j) made by $\mathcal{A}$, $\mathcal{A}'$ replies with $\langle h_2 \cdot g^{r_j}, \mathit{\Pi}_s.\mathrm{Enc}_{k_j}(s_{u_j})\rangle$ where $r_j \leftarrow [q]$, $k_j \leftarrow \{0,1\}^\eta$. $\mathcal{A}'$ also computes and stores $R_j = \left(pk_a^{r_j}\right)^{-1}$.
All random oracles queries are simulated by sampling the output of the oracle u.a.r. for new queries and using the value first sampled for repeated queries. During the simulation $\mathcal{A}'$ also pays attention to the following:
   - If any of the following events occur, $\mathcal{A}'$ aborts the simulation and outputs 1:
     - $\mathcal{A}$ queries H_{DH} for a hidden Diffie-Hellman key on an encryption edge $(u, v) \in E$ with $u \neq a$
     - $\mathcal{A}$ queries H_{gen} or H_{dep} for a hidden seed
     - $\mathcal{A}$ queries corrupt(u) for some node u such that a is no longer safe
   - If $\mathcal{A}$ queries q_i to H_{DH} such that $q_i \cdot R_j = k$ for some j, $\mathcal{A}'$ aborts the simulation and outputs 0. This is the only point at which $\mathcal{A}'$ outputs 0.

   If the simulation arrives to the point where $\mathcal{A}$ outputs its guess (step 4. of the SD-GSD game), then $\mathcal{A}'$ outputs 1.

The advantage of $\mathcal{A}'$ is given by

$$\mathrm{Adv}^{\mathrm{DDH}}_{\mathcal{G},\eta}(\mathcal{A}') \overset{(4)}{=} \Pr[0 \leftarrow \mathcal{A}' \mid b = 0] - \Pr[0 \leftarrow \mathcal{A}' \mid b = 1], \tag{16}$$

where b is the bit sampled by the DDH challenger.

First, we will show that

$$\Pr[0 \leftarrow \mathcal{A}' \mid b = 0] \geq \frac{\Pr[F_{\mathrm{DH}}]}{N}. \tag{17}$$

This part of the proof proceeds very similarly to the proof of Lemma 2 on page 22 and we will be a bit more concise. We focus on executions of $\mathrm{Game}^{\mathrm{DDH}}_{\mathcal{G},\eta}(\mathcal{A}')$ with $b = 0$. Let the games G_1, G_2 be defined as in Lemma 2, where we denote the node sampled at the beginning of each game by a_1, a_2, respectively (as opposed to w_1, w_2). Let $E = F_{\mathrm{DH}}$ and let E_1, E_2 and E' be the analogous events in G_1, G_2 and the game simulated by $\mathcal{A}'$ (note that in this latter game, the group elements $pk_a^{\log_g(h_2)+r_j}$ are also hidden Diffie-Hellman keys). Finally, we introduce the random variable

$$A = \begin{cases} 0 & \overline{F_{\mathrm{DH}}} \\ x & F_{\mathrm{DH}} \text{ holds and } Q_{\mathrm{DH}} \text{ was triggered on an encryption edge with source } x \end{cases}$$

(if x is not unique we choose the node with smallest identifier) and let A_1, A_2 and A' denote the corresponding random variables in game G_1, game G_2 and the game simulated by $\mathcal{A}'$.

Just as argued in Lemma 2,

$$\Pr[E_1] = \Pr[E] \tag{18}$$

holds, since whenever G_1 aborts, it is already decided whether F_{DH} holds:

- If the game was aborted when $\mathcal{A}$ queried a hidden Diffie-Hellman key, then F_{DH} holds.
- If the game was aborted when $\mathcal{A}$ queried H_{gen} or H_{dep} for a hidden seed, F_{DH} does not hold.

Next, the inequality

$$\Pr[A_1 = a_1 \mid E_1] \geq \frac{1}{N}$$

and therefore also

$$\Pr[A_1 = a_1] \geq \frac{1}{N} \cdot \Pr[E] \tag{19}$$

hold for the same reason that

$$\Pr[W_1 = w_1 \mid E_1] \geq \frac{1}{N}$$

and (9) held in Lemma 2.

Then, the equality

$$\Pr[A_1 = a_1] = \Pr[A_2 = a_2] \tag{20}$$

holds again due to the fact that when G_2 aborts because a_2 is no longer safe, we know that $A_2 \neq a_2$.

Finally, we need to argue that

$$\Pr[A_2 = a_2] = \Pr[A' = a]. \tag{21}$$

Consider how G_2 differs from the game simulated by $\mathcal{A}'$. As in Lemma 2, both games abort at exactly the same events (note that if $q_i \cdot R_j = k$ holds and $\mathcal{A}'$ outputs 0, then $q_i = k \cdot R_j^{-1} = k \cdot pk_a^{r_j} = h_1^{\log_g(h_2)} \cdot pk_a^{r_j} = pk_a^{\log_g(h_2)+r_j}$, a hidden Diffie-Hellman key). The game simulated by $\mathcal{A}'$ differs in two aspects:

(i) $\mathcal{A}'$ sets pk_a to h_1 and not to the public key output by $\Pi_{\text{DH}}.\text{Gen}(1^\eta, H_{\text{gen}}(s_a))$
(ii) $\mathcal{A}'$ answers queries $\text{encrypt}(a, u)$ differently

Note that as long as the game G_2 is ongoing, $\mathcal{A}$ has not queried H_{gen} for s_a or H_{DH} for a hidden Diffie-Hellman key. Both differences are therefore indistinguishable:

(i) By Definition 11, the output of $\Pi_{\text{DH}}.\text{Gen}(1^\eta, r)$ with $r \leftarrow \{0,1\}^\rho$ follows the same distribution as the output of $\Pi_{\text{DH}}.\text{Gen}(1^\eta)$. The former process is behind the distribution of pk_a as viewed from $\mathcal{A}$ in G_2, as $\mathcal{A}$ has not queried $H_{\text{gen}}(s_a)$, and the latter process is behind the distribution of pk_a in the game simulated by $\mathcal{A}'$, as the DDH challenger generates a public key with the same distribution as $\Pi_{\text{DH}}.\text{Gen}(1^\eta)$. Since both processes follow the same distribution, pk_a follows the same in G_2 and the game simulated by $\mathcal{A}'$ from $\mathcal{A}$'s perspecive.
(ii) In G_2 a query $\text{encrypt}(a, u)$ is answered with $\langle g^z, c\rangle$ where $z \leftarrow [q], c \leftarrow \Pi_s.\text{Enc}_k(s_u)$ and $k = H_{\text{DH}}(pk_a^z)$. $\mathcal{A}'$ answers such a query with $\langle g^{\log_g(h_1)+r}, c'\rangle$ where $r \leftarrow [q], c' \leftarrow \Pi_s.\text{Enc}_{k'}(s_u)$ and $k' \leftarrow \{0,1\}^\eta$. First, $\log_g(h_1) + r$ follows the same distribution as z. Second, pk_a^z is a hidden Diffie-Hellman key and from $\mathcal{A}$'s view k follows the same distribution as k'.

Thus (21) indeed holds.

Now, again analogous to Lemma 2 if the event $A' = a$ occurred, then $\mathcal{A}'$ outputs 0 and

$$\begin{aligned}\Pr[0 \leftarrow \mathcal{A}' \mid b = 0] \geq \Pr[A' = a]& \\ &\overset{(21)}{=} \Pr[A_2 = a_2] \\ &\overset{(20)}{=} \Pr[A_1 = a_1] \\ &\overset{(19)}{\geq} \frac{\Pr[E]}{N} \\ &= \frac{\Pr[F_{\mathrm{DH}}]}{N}.\end{aligned}$$

Second, we will show that $\Pr[0 \leftarrow \mathcal{A}' \mid b = 1]$ is negligible. When $b = 1$ in $\mathrm{Game}^{\mathrm{DDH}}_{\mathcal{G},\eta}(\mathcal{A}')$, k is a uniformly random group element independent of any information given to $\mathcal{A}$, in particular of $q_i \cdot R_j$ for any i, j. Thus for any i, j,

$$\Pr[q_i \cdot R_j = k] = \frac{1}{q}.$$

Thus, by a union bound and using that $i \in [m_{\mathrm{DH}}], 1 \leq j \leq N - 1 \leq N$ (j is bounded by the maximum outdegree) we have

$$\Pr[0 \leftarrow \mathcal{A}' \mid b = 1] \leq \frac{m_{\mathrm{DH}} \cdot N}{q}. \tag{22}$$

Combining (16), (17) and (22) we get

$$\mathrm{Adv}^{\mathrm{DDH}}_{\mathcal{G},\eta}(\mathcal{A}') \geq \frac{\Pr[F_{\mathrm{DH}}]}{N} - \frac{m_{\mathrm{DH}} \cdot N}{q}. \tag{23}$$

Furthemore, going through the details yields that $\mathcal{A}'$ runs in time

$$\begin{aligned}t_{\mathcal{A}'} \coloneqq \tilde{t} + \mathcal{O}\big(\rho \cdot t_{\mathrm{sample}} \cdot m_{\mathrm{s}} + (\gamma + \eta \cdot t_{\mathrm{sample}}) \cdot m_{\mathrm{DH}}& \\ &+ N \cdot ((\rho + \eta) \cdot t_{\mathrm{sample}} + m_{\mathrm{DH}} \cdot t_{\mathrm{op}} + t_{\Pi_{\mathrm{DH}}.\mathrm{Gen}}) \\ &+ N^2 \cdot t_{\Pi_{\mathrm{DH}}.\mathrm{Enc}}\big).\end{aligned}$$

Then using the definition of $\tilde{t}$, with appropriately chosen constants we have $t_{\mathcal{A}'} \leq t$. So by virtue of the DDH problem being $(t, \varepsilon_{\mathrm{DDH}})$-hard relative to $\mathcal{G}$

$$\mathrm{Adv}^{\mathrm{DDH}}_{\mathcal{G},\eta}(\mathcal{A}') \leq \varepsilon_{\mathrm{DDH}}$$

and if we combine this with (23) we get

$$\begin{gathered}\frac{\Pr[F_{\mathrm{DH}}]}{N} - \frac{m_{\mathrm{DH}} \cdot N}{q} \leq \varepsilon_{\mathrm{DDH}} \\ \iff \\ \Pr[F_{\mathrm{DH}}] \leq N \cdot \varepsilon_{\mathrm{DDH}} + \frac{m_{\mathrm{DH}} \cdot N^2}{q},\end{gathered}$$

concluding the proof.

C TreeKEM security

C.1 Continuous Group Key Agreement

The model As already described briefly in the introduction, a CGKA scheme allows a group of users to agree on a group key, indistinguishable from random for any eavesdropper, while providing mechanisms to add or remove users from the group and update the group key and users' key material, such that FS and PCS can be achieved.

Fully modelling a group of users running a CGKA scheme is complex. Since the protocol must work in the asynchronous setting, there must be a delivery service that takes protocol messages and forwards them to the recipients. Users also need to be able to publish some kind of public key, the key packages used in TreeKEM, such that they can be invited to the group with a welcome message. This functionality is also left to the delivery service. Moreover, there must be mechanisms in place to authenticate protocol messages and the published public keys.

In our model, users are honest nodes running the protocol algorithms and maintaining local state. They send out new messages and process received messages immediately. They have a reliable communication channel to the delivery service, and all public keys and protocol messages are assumed to be authenticated, meaning that an attacker cannot forge them. The delivery service, and thus an attacker, can of course see all protocol messages. We assume little about what messages get delivered by the delivery service: the service may deliver a message to some users but not others and it may not deliver certain messages at all.

For a more complete model we refer the reader to [3]. The authors consider not just CGKA but the more difficult problem of *secure group messaging* as tackled by the MLS protocol. The model they consider allows an attacker to inject protocol messages and gives them some control over the public keys stored by the delivery service.

PC-CGKA schemes Multiple definitions of the syntax and security of CGKA schemes already exist [2,1,3], all meant to capture the syntax of how update, add and remove operations were performed with the latest verion of TreeKEM at the time, and all with the same name. As already described in the introduction, the current version of TreeKEM uses propose and commit operations to advance the group state, which is also the syntax formalized in [3] and in this work. The syntax defined in [2,1] came before the propose and commit syntax was introduced. In this syntax there are no proposals and every operation is a commit, either adding a single user, removing a single group member, or just updating the key material of the committer. To differentiate our definitions from existing ones that describe something different as in [2,1], we will talk about *propose and commit continuous group key agreement* (PC-CGKA) schemes.

Syntax Our definition of the syntax of PC-CGKA schemes is inspired from the definition in [1] and is essentially the same as what is described in [3, Section 4.1.1]. We assume that every user u is identified by some value id_u.

Definition 17 (PC-CGKA). *Let η denote the security parameter. A* PC-CGKA scheme Σ *with key space* $\mathcal{K}$ *consists of the following algorithms:*

INITIALIZATION:

- *An algorithm* Gen*. Before joining any group, a user generates a pair of keys* $(pk, sk) \leftarrow \mathrm{Gen}(1^\eta)$*, a public and private key.*
- *An algorithm* CreateGroup*. A user runs* $\sigma \leftarrow \mathrm{CreateGroup}(1^\eta)$ *to locally initialize a group with themselves as the only member and the state of the group stored in* σ*. We call* σ *their* group state*.*

COMPUTE THE GROUP KEY:

- *An algorithm* Key*. At any point in time, a member of a group with state* σ *can compute the current* group key $k \leftarrow \mathrm{Key}(\sigma)$ *with* $k \in \mathcal{K}(\eta)$*.*

PROPOSAL:

- *An algorithm* ProposeUpdate*. If a member* u *of a group with state* σ *wishes to update their key material, they may run* $(\sigma, p) \leftarrow \mathrm{ProposeUpdate}(\sigma)$ *to create an* update proposal p *to be shared with other members of the group and update their state such that they have processed* p*.*
- *An algorithm* ProposeAdd*. If a member of a group with state* σ *wishes to add a new user* u *with public key* pk_u *to the group, they may run* $(\sigma, p) \leftarrow \mathrm{ProposeAdd}(\sigma, id_u, pk_u)$ *to create an* add proposal p *to be shared with other members of the group and update their state such that they have processed* p*.*
- *An algorithm* ProposeRemove*. If a member of a group with state* σ *wishes to remove another member* u *from the group, they may run* $(\sigma, p) \leftarrow \mathrm{ProposeAdd}(\sigma, id_u)$ *to create a* remove proposal p *to be shared with other members of the group and update their state such that they have processed* p*.*

COMMIT:

- *An algorithm* CreateCommit*. To apply a list of proposals* π *to the group state, a member with state* σ *may run* $(\sigma', c, w_1, \ldots, w_k) \leftarrow \mathrm{CreateCommit}(\sigma, \pi)$*, where* c *is a* commit *to be shared with other members,* σ' *would be the new state of the member after applying the commit*[7] *and each* w_i *is a* welcome message *for a newly added user.*

PROCESS:

- *An algorithm* ProcessCommit*. Upon receiving another member's commit* c*, a member* u *with state* σ *can set* $\sigma \leftarrow \mathrm{ProcessCommit}(\sigma, c)$ *to process* c*. We say that* u *has* processed c*.*
- *An algorithm* ProcessWelcome*. Upon receving a welcome message* w *for a user with public key* pk*, the user with this public key can set* $\sigma \leftarrow \mathrm{ProcessWelcome}(pk, sk, w)$*, where* sk *is the corresponding secret key output by* Gen*.*

For any object X *above (including* $\mathcal{K}$*) we will refer to it as* $\Sigma.X$*.*

The scheme must also specify an algorithm for determining the set of members of the group from a group state σ*.*

[7] Note that the user's state is not immediately replaced with the new state output by the algorithm. We will see why in the explanation of the semantics below.

Semantics In the following we provide some further details regarding the semantics of the PC-CGKA algorithms:

- Gen: The public key is used to invite the user to the group and should therefore be made public. This public key corresponds to a key package in TreeKEM (see Section 1.2). The same key pair must not be reused to join multiple groups and must be discarded after it was used to join a group. A new key pair must be generated to join a new group.
- ProposeUpdate: An update proposal created by a user u contains (possibly public) information for the other group members about u's new key material. This information is used by other members to provide encrypted information in a commit (see below) that includes the update proposal such that u is able to compute the new group key.
- CreateCommit: Let c a commit and $w_1, \ldots, w_k$ the corresponding welcome messages output by the algorithm, run by user u with group state σ and with the proposals π provided as input. There should be one welcome message for each new user added to the group in the commit with a corresponding add proposal in π. Welcome message w_i contains the identifier id_i of a user and the message should be shared with that user such that they can join the group. Besides updating the key material for all other members with an update proposal in π, the commit also updates user u's key material. Accordingly, π should not contain an update proposal for user u. Nor should it contain a remove proposal for user u as they will know the group key resulting from the commit. User u may keep both group states σ and σ' until the group agrees on whether to apply the commit c or not. If the commit is to be applied, user u sets their state to σ' and discards σ. Otherwise, they discard σ'. Applying a commit results in a new group key.

  Our syntax does not specify how a user learns of proposals in π created by other users. Also how users agree on whether to apply a commit is left up to the application. The decision could be made by the delivery service or using some consensus algorithm run by all group members.

  We see a call to CreateGroup as a special type of commit that is applied by the group creator.
- ProcessCommit: If the commit removed the member from the group, they should not be able to compute the group key from σ and should delete σ.
- ProcessWelcome: The user must discard their secret key sk after processing a welcome message, so that the contents of the welcome message remain secret in case the user gets compromised (recall FS). As we cannot express this conveniently with our syntax, our security definition does not check for this (and does not give the adversary the secret key if a user is compromised after having processed their welcome message).

Correctness The above description of semantics already provides some explicit correctness properties or implicitly implies other ones. We will explicitly define one important correctness property that a PC-CGKA scheme should satisfy in Definition 21 on page 43. **TODO:** Justify why I mention this property and why it is important. Explain examples of commit history and why it plays a role in security.

The correctness property concerns the handling of "bad" (malformed or inconsistent) inputs. The algorithms of a PC-CGKA scheme should have several checks built in to deal with such inputs. For example

- a commit including an update or add proposal for the commit creator is invalid
- a user should never process the same commit twice
- a user should never process a commit that they created
- etc.

Many of these checks are straightforward and we do not provide an extensive list of what is needed. However, we will discuss one type of check that is less straightforward and plays a role in the security of the scheme. Our correctness property enforces all members of a group to agree on the history of commits they have applied (up to joining the group). It avoids scenarios where a group member may skip a commit processed by other members that, for example, removed a user from the group. We ignore errors that would result from processing bad input in our syntax and restrict our security model to dealing with only valid inputs, as it is not our goal to analyze this type of attack on the scheme.

Before we can formally define our correctness property, we must first introduce some definitions.

Definition 18 (Applying a commit). *When a user*

- *processes commit c with* ProcessCommit
- *creates commit c and subsequently updates their group state to the new state output by the corresponding call to* CreateCommit
- *joins the group by processing welcome message w, where c is the commit that was output along with w by* CreateCommit
- *creates a group, where we let c denote the call to* CreateGroup

we say that the user applied *commit c.*

In the following, when talking about *time* for a user that was a part of some group, we are referring to the sequence of group states they went through as members of the group[8].

Definition 19 (Last commit). *Let u a user that at some point in time was a member of a group and had group state σ. We define the* last commit *in σ to be the most recent commit c that u applied up to arriving in state σ.*

In the above definition, the user's last commit will always exist since they either joined the group through a welcome message or created the group themselves.

[8] We are only interested in state transitions from applying a commit, but for completeness we will also consider transitions due to creating proposals as a part of this sequence.

Definition 20 (Consistent group states). *Let u_0, u_1 two users where each user was a member of a group at some point in time. Let σ_0, σ_1 the group states they were in, respectively and c_0, c_1 the last commits in σ_0, σ_1, respectively. The group states σ_0 and σ_1 are said to be* consistent *if $c_0 = c_1$.*[9]

We can now define the correctness property motivated above.

Definition 21 (Consistent history). *A PC-CGKA scheme Σ* maintains a consistent hisory *if a user with group state σ only successfully*[10] *processes a commit $c \leftarrow \text{CreateCommit}(\sigma', \cdot)$ for some σ'*[11] *(with* ProcessCommit*) if σ and σ' are consistent.*

Definition 20 also allows us to express the following important correctness property: any set of members with consistent group states must compute the same group key with $\Sigma.\text{Key}$ and must agree on the set of members of the group.

In the following we introduce a few more definitions that will become useful later.

Definition 22 (Parent commit). *Let c a commit output by* $\text{CreateCommit}(\sigma_0, \cdot)$ *for some σ_0. The* parent commit *of c is the last commit in σ_0.*

Note that if the PC-CGKA scheme maintains a consistent history, for a commit c that was processed by a user while they were still in group state σ, the last commit in σ will be the parent commit of c.

Definition 23 (Commit history). *Let c a commit.*[12] *Define the* commit history *of c as follows:*

- ***Case** c **refers to a call to*** CreateGroup*: the sequence (c) of length 1*
- ***Otherwise:*** *the sequence $(c_1, \ldots, c_k, c)$, where $(c_1, \ldots, c_k)$ is the commit history of c's parent commit c_k.*

One could also consider the *local* commit history of a group member u in group state σ, consisting of the sequence of commits applied by u since joining the group and until arriving in σ. If the PC-CGKA scheme maintains a consistent history, this local commit history is a suffix of the commit history of the last commit in σ. (To see this, first note that by definition the last commit in σ is

[9] If a commit is a call to CreateGroup, it is equal to another commit iff. both refer to the same call to CreateGroup. This implies that after a user just created a group, their group state is consistent with itself only.

[10] As noted, we ignore checks for bad input in our syntax. To describe schemes satisfying correctness related to bad inputs, one would need to extend the syntax such that e.g. an algorithm can also output an error, and the user's state remains unchanged if this is the case.

[11] Here we only consider states σ' that an honest user would get as output from one of the PC-CGKA algorithms.

[12] Here we only consider commits referring to a call to CreateGroup or output by CreateCommit, run by an honest user.

the last commit the local commit history. Then repeatedly apply the argument before Definition 23.) Thus, for a set of users in consistent group states, the users all agree on the commits they have processed and their order (up to the earliest commit present in the local commit history of all users).

PC-CGKA security Our security definition is again inspired by [1]. We consider fully adaptive adversaries. The adversary controls all PC-CGKA operations performed by the users, can decide who receives what messages (i.e. the adversary has control over the delivery service), can decide what commits get applied or discarded, and when they are discarded, by querying "confirm" and can corrupt the state of any user. We will refer to commits created by a user that they have not yet been told to apply or discard as *unconfirmed commits*. Corrupting a user leaks the group states corresponding to all their unconfirmed commits. Because the adversary can schedule the delivery of messages as it likes, it is possible for the adversary to create "forks" in the group where some users in the group are told to process one commit, while other users are told to process another. Such forks could also happen in practice and should not break security.

The adversary eventually chooses a commit to be challenged on, for which they must differentiate the group key from a uniformly random key. We must restrict the set of commits the adversary can ask to be challenged on to those that are expected to be *safe* even in the face of previous or later corruptions. The level of FS and PCS expected from a PC-CGKA scheme is captured by the size of this set of safe commits. Exactly which commits are considered safe will be explained later.

We also impose some notable restrictions on the adversary. The adversary cannot inject protocol messages or public keys and it may only deliver a message to users that are supposed to process that message, in order to avoid giving users messages with bad inputs. The latter restriction is justified, as in a correct PC-CGKA protocol such messages would simply be discarded and correctness can be verified independently. Imposing the restriction on the adversary allows us to ignore the details of handling bad inputs when specifying a PC-CGKA scheme and to analyze the core aspects of its security.

The definition in [3, Section B.1] is very similar in essence. The same restricions on the adversary are imposed. However, the security game provided there gives more power to the adversary: the adversary may additionally choose the randomness used in operations, choose its own public keys to be associated with users and tell certain users not to delete old keys. In the end this restricts the set of safe commits. We provide our own definition with the hope of having a formulation that is easier to digest, keeps the security game simpler and is more explicit about what commits are considered safe.

Definition 24 (The PC-CGKA game). *Let Σ a PC-CGKA scheme. Define the game* $\mathrm{Game}^{\mathrm{PC-CGKA}}_{\Sigma,\eta}(\mathcal{A})$ *for an adversary* $\mathcal{A}$*:*

1. *$\mathcal{A}$ outputs $n \in \mathbb{N}$. For each $i \in [n]$, initialize a user i by creating a (unique) identifier id_i, generating $(pk_i, sk_i) \leftarrow \Sigma.\mathrm{Gen}(1^\eta)$, preparing $U_i = \varnothing$, the set*

of unconfirmed commits at user i, and setting $\sigma_i \coloneqq \varnothing$, where $\varnothing$ denotes the empty value. The state output by an algorithm of Σ is never the empty value.
$\mathcal{A}$ is given $(pk_1, id_1), \ldots, (pk_n, id_n)$.
Set $P = C = W = 0$, where P denotes the number of proposals, C the number of commits and W the number of welcome messages created.

2. $\mathcal{A}$ may adaptively do the following queries:
   - create-group(i) for $i \in [n]$: set $\sigma_i \leftarrow \mathrm{CreateGroup}(1^\eta)$.
   - propose-update(i) for $i \in [n], \sigma_i \neq \varnothing$: run $(\sigma_i, p_{P+1}) \leftarrow \mathrm{ProposeUpdate}(\sigma_i)$ to update user i's state and get a proposal p_{P+1}. $\mathcal{A}$ is given p_{P+1}. Set $P \coloneqq P + 1$.
   - propose-add(i, j) for $i, j \in [n], \sigma_i \neq \varnothing$: run $(\sigma_i, p_{P+1}) \leftarrow \mathrm{ProposeAdd}(\sigma_i, id_j, pk_j)$ to update user i's state and get a proposal p_{P+1}. $\mathcal{A}$ is given p_{P+1}. Set $P \coloneqq P + 1$.
   - propose-remove(i, j) for $i, j \in [n], \sigma_i \neq \varnothing$: run $(\sigma_i, p_{P+1}) \leftarrow \mathrm{ProposeRemove}(\sigma_i, id_j)$ to update user i's state and get a proposal p_{P+1}. $\mathcal{A}$ is given p_{P+1}. Set $P \coloneqq P + 1$.
   - create-commit$(i, (j_1, \ldots, j_d))$ for $i \in [n], \sigma_i \neq \varnothing, \forall l\ j_l \in [P]$: run $(\sigma, c_{C+1}, w_{W+1}, \ldots, w_{W+k}) \leftarrow$ $\mathrm{CreateCommit}(\sigma_i, (p_{j_1}, \ldots, p_{j_d}))$ to create the new state σ, commit c_{C+1} and corresponding welcome messages. $\mathcal{A}$ is given c_{C+1} and $w_{W+1}, \ldots, w_{W+k}$. Set $U_i \coloneqq U_i \cup \{(C+1, \sigma)\}$, $C \coloneqq C + 1$ and $W \coloneqq W + k$.
   - confirm(j, b) for j s.t. $(j, \sigma) \in U_i$ for some user i and state σ, $b \in \{0, 1\}$: If $b = 0$, set $U_i \coloneqq U_i \setminus \{(j, \sigma)\}$. If $b = 1$, set $\sigma_i \coloneqq \sigma$ and $U_i \coloneqq \varnothing$.[13]
   - deliver-commit(i, j) for $i \in [n], \sigma_i \neq \varnothing, j \in [C]$: run $\sigma \leftarrow \mathrm{ProcessCommit}(\sigma_i, c_j)$. Set $U_i \coloneqq \varnothing$. If c_j contains a remove proposal for user i, then set $\sigma_i \coloneqq \varnothing$, generate a new pair $(pk_i, sk_i) \leftarrow \Sigma.\mathrm{Gen}(1^\eta)$ and give (i, pk_i) to $\mathcal{A}$. Otherwise, set $\sigma_i \coloneqq \sigma$.
   - deliver-welcome(i, j) for $i \in [n], \sigma_i = \varnothing, j \in [W]$: set $\sigma_i \leftarrow \mathrm{ProcessWelcome}(pk_j, sk_j, w_j)$.[14]
   - corrupt(i) for $i \in [n]$: If $\sigma_i = \varnothing$, $\mathcal{A}$ is given sk_i. Otherwise, $\mathcal{A}$ is given σ_i and U_i.
3. $\mathcal{A}$ picks $i \in [0, C]$. We call the commit c_i the challenge commit, where c_0 refers to the initial CreateGroup operation. Let σ the group state output by the operation that created c_i (the state output by CreateCommit if $i > 0$ or the state output by CreateGroup if $i = 0$). A bit $b \leftarrow \{0, 1\}$ is sampled and $\mathcal{A}$ is given
$$k = \begin{cases} \Sigma.\,\mathrm{Key}(\sigma) & b = 0 \\ \tilde{k} & b = 1 \end{cases},$$
where $\tilde{k} \leftarrow \Sigma.\mathcal{K}(\eta)$. $\mathcal{A}$ may continue to do queries as before.
4. $\mathcal{A}$ outputs a bit b'. The output of the game is defined to be 1 if $b' = b$, and 0 otherwise.

[13] All other unconfirmed commits in U_i are cleared if $b = 1$ as they should not be applied anymore.

[14] Note that in a real execution of the protocol the user must delete sk_j from their local state after processing the welcome message w_j. Accordingly, sk_j is no longer leaked to the adversary in a later query corrupt(j).

We require an adversary playing the above game to adhere to the following:

- create-group *is queried exactly once*
- *The challenge commit is safe (see Definition 28 on page 48)*
- *For any query* deliver-commit(i, j) *where the commit c_j was created by user k while they where in state σ'_k, σ_i and σ'_k must be consistent*
- *For any query* create-commit$(i, (j_1, \ldots, j_d))$*, for every proposal p_{j_l} created by a user while in state σ'_l, σ_i and σ'_l must be consistent*
- *A user never processes a commit that they created*
- *Every commit is processed at most once by any user*
- *A welcome message for user i is processed by i at most once and is never processed by a user j with $i \neq j$*
- *A user creating a commit never includes an update or remove proposal for themselves, or multiple update/add/remove proposals for to the same user*
- *A user is never asked to create an add proposal for a user they consider to be in the group, or create a remove proposal for a user they do not consider to be in the group*

The concept of a safe user and safe commit is adapted from the so-called "safe predicate" in [1], which again took inspiration from [2]. As elaborated in the cited papers and also analogous to how we needed to define "safe" nodes in the SD-GSD game, we want to forbid the adversary to ask to be challenged on a commit for which it can trivially compute the group key through some corruption it performed.

To see what is needed for a commit to be safe, consider some commit c with group key k created by a user i and let $j \neq i$ any user that i would consider to be in the group after applying c (Definition 25 clarifies exactly which users are considered to be in the group). The commit c or an associated welcome message provides encrypted information for user j to compute the new group key using its current key material. Clearly, if this key material has been compromised by the adversary corrupting user j, the commit should not be safe. If the adversary has not corrupted user j since they last updated their key material, then we would not expect the adversary to be able to learn the group key k through user j, even if user j was corrupted before (recall PCS). Moreover, corrupting user j after they have again updated their key material should not allow the adversary to compute the group key of c either (recall FS). We will later say that the commit c is *safe with respect to user j* if j was not corrupted in this window between their last and next update. Now, it is important to notice that the encrypted information in commit c is for the key material that user j had *from user i's view* when user i created c. It is possible that when user i created c, user j had already processed a commit updating their key material that user i has not yet processed. Thus, we must be careful to require exactly the right key material of user j to be unknown to the adversary. Definition 27 on the next page formalizes this.

Definition 25. *Let c a commit and let σ' the new group state ouptut by*

- *the call to* CreateCommit *that created c*

- *or the call to* CreateGroup *that c refers to*

The (set of) users in the group after applying c *is the set of users in the group according to state σ'.*

Definition 26. *Let c a commit and let u a user in the group after applying c. Let $h = (c_1, \ldots, c_k)$ the commit history of c. Define u's* last update up to c *as the last commit c_i to satisfy one of the following:*

(i) *c_i was created by u*
(ii) *c_i included an update proposal for u*
(iii) *c_i was output along with a welcome message for u*
(iv) *c_i refers to a call to* CreateGroup *run by u (implying $i = 1$)*

Definition 27 (Safe user). *Let Σ a PC-CGKA scheme and let η arbitrary. Consider an execution of $\mathrm{Game}^{\mathrm{PC-CGKA}}_{\Sigma,\eta}(\mathcal{A})$ for some adversary $\mathcal{A}$. Let Q the total number of queries made by $\mathcal{A}$. We will refer to queries by their index among all queries. Let $q^* \in [Q]$ a* create-group(i) *or* create-commit$(i, \cdot)$ *query with $i \in [n]$ as the target user. Let $j \in [n]$ any user (including i) in the group after applying the commit c^* created by q^*. Let the commit c' be user j's last update up to c^*.*

Set the query $q^- \in [Q]$ depending on which case in Definition 26 commit c' falls into:

(i) *q^- is the* create-commit$(j, \cdot)$ *query that created c'*
(ii) *q^- is the* propose-update(j) *query that created the update proposal for j included in c'*
(iii) *Let q_{add} be the query to* propose-add *that created the add proposal for user j that was included in c'. q^- is the last* deliver-commit *query before q_{add} that reset j's public and private key pair, or set $q^- = 0$ if no such query was made.*
(iv) *q^- is the corresponding query* create-group(j) *that ran* CreateGroup

Analogously, set the query $q^+ \in [Q]$ depending on which case in Definition 26 commit c' falls into:

(i)
  - ***Case user j applied** c': Then a query* confirm$(k, 1)$ *with index q_{confirm} was made where $c_k = c'$. Set q^+ same as in (iv), but with $q^+ > q_{\mathrm{confirm}}$.*
  - ***Otherwise:** q^+ is the next query that removed the new state associated with c' from U_j. This is either a query* confirm$(k, 0)$ *with $c_k = c'$, a query* confirm$(k, 1)$ *with $c_k \neq c'$ or a query* deliver-commit(j, k) *with $c_k \neq c'$. Set $q^+ = Q$ if there is no such query.*

(ii) *Let p be the update proposal for j included in c'.*
  - ***Case j applied a commit c_k that included** p: same as (iv), but use the next query after q_{deliver}, where q_{deliver} is the* deliver-commit(j, k) *query that let j process c_k*
  - ***Otherwise:** q^+ is first query such that $q^+ > q^-$ that led to user j applying a commit, or set $q^+ = Q$ otherwise*[15]

[15] Once user j has applied any commit after creating the update proposal p that does not include p, it is clear that p is outdated and its associated data should be deleted.

(iii) *same as* (*iv*)

(iv) q^+ *is first query such that* $q^+ > q^-$ *that led to user* j *applying a commit* c *that they created (so* q^+ *is a* confirm$(k, 1)$ *query with* $c_k = c$*) or that included an update or remove proposal for* j *(so* q^+ *is a* deliver-commit(j, k) *query with* $c_k = c$*), or set* $q^+ = Q$ *if no such commit exists*[16]

The commit c^* *is* safe with respect to user j *if there was no* corrupt(j) *query in the interval* $[q^-, q^+]$.

TODO: Explain how this definition improves on [1]. ([1] is ambiguous in some case and also makes the interval unnecessarily large.)

Continuing the discussion above, so far we have considered a necessary condition to keep the commit c safe by restricting the corruptions made to a specific user j. If c is safe with respect to every user that i considered to be in the group after applying c (including user i), we would expect that the adversary is not able to compute the corresponding group key. Indeed, this is how we define a safe commit.

Definition 28 (Safe commit). *Recall the setting of Definition 27. As in Definition 27, let* $q^* \in [Q]$ *a* create-group(i) *or* create-commit$(i, \cdot)$ *query with* $i \in [n]$ *as the target user and let* c^* *the commit created by* q^**. The commit* c^* *is* safe *if for every user* j *(including* i*) in the group after applying commit* c^**, the commit* c^* *is safe with respect to user* j*.*

Definition 29 (PC-CGKA security). *A PC-CGKA scheme is* $(t, \varepsilon, c, p, u)$-PC-CGKA-secure *if for all* η*, for any adversary* $\mathcal{A}$ *making at most* $c(\eta)$ *queries to* create-commit*, creating at most* $p(\eta)$ *update or add proposals in the created commits and asking for at most* $u(\eta)$ *users in step 1. of the PC-CGKA game we have*

$$\mathrm{Adv}_{\Sigma,\eta}^{\mathrm{PC-CGKA}}(\mathcal{A}) := 2 \cdot \left(\Pr\left[\mathrm{Game}_{\Sigma,\eta}^{\mathrm{PC-CGKA}}(\mathcal{A}) = 1\right] - \frac{1}{2}\right) \leq \varepsilon(\eta).$$

C.2 The TreeKEM Protocol

The TreeKEM protocol discussed in the literature is not described as a self-contained subprotocol in the MLS specification [6] and is therefore only defined implicitly. The following description of the protocol was extracted from [6]. Fully describing TreeKEM is complex and some parts of the protocol were either simplified (e.g. the content of protocol messages) or omitted as they are not relevant for proving security with respect to our definition (e.g. handling of bad inputs, signatures, hashes of the tree and additional functionality provided by the protocol).

[16] This is simply describing the next query that made user j update their key material, and therefore delete their old key material.

Definition 30 (TreeKEM [6]). *Let Π a public-key encryption scheme, where $\Pi.\mathrm{Gen}(1^\eta)$ uses $\rho(\eta)$ bits of randomness. Let $\mathcal{H}_{\mathrm{gen}} = \{H^{(\eta)}_{\mathrm{gen}} \mid \eta \in \mathbb{N}\}, \mathcal{H}_{\mathrm{dep}} = \{H^{(\eta)}_{\mathrm{dep}} \mid \eta \in \mathbb{N}\}$ families of functions with $H^{(\eta)}_{\mathrm{gen}}, H^{(\eta)}_{\mathrm{dep}} \colon \{0,1\}^{\rho(\eta)} \to \{0,1\}^{\rho(\eta)}$. We write $H_{\mathrm{gen}} \coloneqq H^{(\eta)}_{\mathrm{gen}}, H_{\mathrm{dep}} \coloneqq H^{(\eta)}_{\mathrm{dep}}$ and $\rho \coloneqq \rho(\eta)$ if η is clear from the context. Define the CGKA scheme Σ_{TK} with key space $\mathcal{K}(\eta) = \{0,1\}^{\rho(\eta)}$ and its algorithms defined as follows, where id refers to the identifier of the user running the algorithm:*

- Gen*:*
  - *generate $(pk_{\mathrm{init}}, sk_{\mathrm{init}}) \leftarrow \Pi.\mathrm{Gen}(1^\eta)$, where pk_{init} is the init key*
  - *generate the key pair of the user's leaf $(pk_{\mathrm{leaf}}, sk_{\mathrm{leaf}}) \leftarrow \Pi.\mathrm{Gen}(1^\eta)$*
  - *set $pk \coloneqq (pk_{\mathrm{init}}, pk_{\mathrm{leaf}})$, this is the user's key package, and $sk \coloneqq (sk_{\mathrm{init}}, sk_{\mathrm{leaf}})$, this will be stored by the user, and output the key pair (pk, sk)*
- CreateGroup(1^η)*:*
  - *generate $(pk_{\mathrm{leaf}}, sk_{\mathrm{leaf}}) \leftarrow \Pi.\mathrm{Gen}(1^\eta)$*
  - *create a tree with a single node v and set $(pk_v, sk_v) = (pk_{\mathrm{leaf}}, sk_{\mathrm{leaf}})$*
  - *set the group key to $k \leftarrow \{0,1\}^\rho$*
  - *output a state σ containing the tree, the group key k and the security parameter η*
- Key(σ)*: output the group key stored in σ*
- ProposeUpdate(σ)*:*
  - *generate $(pk_{\mathrm{leaf}}, sk_{\mathrm{leaf}}) \leftarrow \Pi.\mathrm{Gen}(1^\eta)$*
  - *create the add proposal $p \coloneqq (\mathrm{update}, id, pk_{\mathrm{leaf}})$ and store sk_{leaf} in σ*
  - *output (σ, p)*
- ProposeAdd(σ, id', pk')*:*
  - *create the add proposal $p \coloneqq (\mathrm{add}, id', pk')$*
  - *output (σ, p)*
- ProposeRemove(σ, id')*:*
  - *create the remove proposal $p \coloneqq (\mathrm{remove}, id')$*
  - *output (σ, p)*
- CreateCommit$(\sigma, (p_1, \ldots, p_k))$*:*
  - *create a commit object c storing all proposals and the author id of the commit*
  - *for every update proposal $p_j = (\mathrm{update}, id', pk')$:*
    - *replace the leaf of user id' with a new leaf with public key pk'*
    - *replace all nodes on the direct path of the new leaf with blank ones*
  - *for every remove proposal $p_j = (\mathrm{remove}, id')$:*
    - *replace the leaf of user id' and all nodes on their direct path with blank nodes*
    - *as long as the right child of the root has an empty resolution (and the root actually has a right child), truncate the tree by deleting the subtree of the root's right child and the root itself, and setting the root's left child as the new root*
  - *for every add proposal $p_j = (\mathrm{add}, id', (pk'_{\mathrm{init}}, pk'_{\mathrm{leaf}}))$ (in order):*

    - *if there are no blank leaves in the tree, extend the tree to the right by setting the root to be a new blank node, the left child of the root to the old root and the right child of the root to a full subtree of blank nodes (of the same height as the old root's subtree)*
    - *replace the leftmost blank leaf in the tree with a new leaf with public key* pk'_{leaf}
    - *for every non-blank node on the new leaf's direct path, add the new leaf to the node's set of unmerged leaves*
  - *generate* $(pk_{\text{leaf}}, sk_{\text{leaf}}) \leftarrow \Pi.\text{Gen}(1^\eta)$ *and sample* $s_1 \leftarrow \{0,1\}^\rho$
  - *replace id's leaf with a new leaf with key pair* $(pk_{\text{leaf}}, sk_{\text{leaf}})$
  - *If the tree consists of a single leaf, set the group key to* s_1*. Otherwise, for the i-th node* v_i *on id's direct path where its child* w_i *on the copath of id has a non-empty resolution:*
    - *if* $i > 1$, *compute* $s_i \coloneqq H_{\text{dep}}(s_{i-1})$ *and* $(pk, sk) = \Pi.\text{Gen}(1^\eta, H_{\text{gen}}(s_i))$
    - *replace* v_i *with a new node* v'_i *with key pair* (pk, sk) *(and no unmerged leaves)*
    - *for every node* u *in the resolution of* w_i*: If* u *is the leaf of a user* id' *and the commit contains an add proposal* $(\text{add}, id', (pk'_{\text{init}}, pk'_{\text{leaf}}))$, *compute a ciphertext* $c_u \leftarrow \Pi.\text{Enc}_{pk'_{\text{init}}}(s_i)$ [17]*. Otherwise, compute a ciphertext* $c_u \leftarrow \Pi.\text{Enc}_{pk_u}(s_i)$ *and store it in the commit* c*.*

    Set the group key to $H_{\text{dep}}(s_d)$ *where* v_d *is the last node on id's direct path, i.e. the root. Store the list of pubic keys* $(pk_{\text{leaf}}, pk_{v'_1}, \ldots, pk_{v'_d})$ *in* c*.*
  - *for every add proposal* $p_j = (\text{add}, id', (pk'_{\text{init}}, pk'_{\text{leaf}}))$*:*
    - *let* l *be the leaf of user* id' *in the tree*
    - *create a welcome message* $w_{id'}$ *containing the identifier* id'*, the ciphertext* c_l *computed above and a copy of the public part of the tree (i.e. the tree without any secret keys)*
  - *output* (σ', c, ω)*, where* σ' *is the new group state of* id' *after applying the above changes to the tree and setting the new group key, and* ω *is the list of welcome messages computed (in any order)*
- ProcessCommit(σ, c)*:*
  - *apply all proposals in* c *to the tree as in* CreateCommit
  - *replace the committer's leaf and the nodes on the committer's (non-blank) direct path with the new nodes created in the commit*[18]
  - *find the right ciphertext* c_u *encrypting the seed of the new node* u *on id's direct path, decrypt it (using the appropriate secret key known to id) and compute (and store) the secret key of* u*, the non-blank nodes above* u *and the new group key (using the same computations involving* H_{dep} *and* H_{gen} *as in* CreateCommit*)*

[17] As defined here, there is no use for the init key in the protocol and we could simply encrypt the seed under the leaf's public key (in other words set $(pk'_{\text{init}}, sk'_{\text{init}}) = (pk'_{\text{leaf}}, sk'_{\text{leaf}})$). In the real TreeKEM protocol the message encrypted using the init key includes additional information and is different from the type of message encrypted under a leaf's public key.

[18] Recall that c contains the public key of each new node.

- *output the updated group state σ' of id containing the new tree and group key*

– $\mathrm{ProcessWelcome}((pk_{\mathrm{init}}, pk_{\mathrm{leaf}}), (sk_{\mathrm{init}}, sk_{\mathrm{leaf}}), w)$*:*
  - *compute the seed $s = \Pi.\mathrm{Dec}_{sk_{\mathrm{init}}}(c)$ of node u in the tree provided in w, where c is the ciphertext provided in w*
  - *compute the secret key of u and the non-blank nodes above u, and store them in the tree, and compute the group key*
  - *output a group state σ containing the tree, the group key and the security parameter η (derived from pk_{init})*

The scheme Σ_{TK} is called the TreeKEM protocol*.*

The full TreeKEM protocol as described in the RFC achieves the correctness property in Definition 21 using a hash of the tree.

C.3 TreeKEM security from SD-GSD security

We have already described the relationship between the TreeKEM protocol and the SD-GSD security game at the beginning of Chapter 4. The following theorem formalizes this.

Theorem 3. *Let Σ_{TK} the TreeKEM protocol instantiated with a public-key encryption scheme Π. Let c, p, u functions in η. Set $N := c \cdot (\lceil \log(u) \rceil + 1) + u + p$ and $\delta := u$. If Π is $(t, \varepsilon, N, \delta)$-SD-GSD-secure in the ROM and the functions $H_{\mathrm{gen}}, H_{\mathrm{dep}}$ in Σ_{TK} are modelled as random oracles, then Σ_{TK} is $(\tilde{t}, \varepsilon, c, p, u)$-PC-CGKA-secure with $\tilde{t} \approx t$.*

Intuition The approach for the proof is straightforward. Given an adversary $\mathcal{A}$ against TreeKEM, we want to construct an SD-GSD adversary $\mathcal{A}'$ that simulates $\mathrm{Game}^{\mathrm{PC-CGKA}}_{\Sigma_{\mathrm{TK}},\eta}$ to $\mathcal{A}$ and uses $\mathcal{A}$'s ability to distinguish the group key of a safe commit from a random key to win the SD-GSD game. Every non-blank node in TreeKEM can be simulated with a corresponding node in the GSD graph. Note that the group key of a commit in TreeKEM is given by $H_{\mathrm{dep}}(s)$ where s is the seed of the root node. Thus, if $\mathcal{A}$ can distinguish the group key of a safe commit from a uniformly random key $k \leftarrow \{0,1\}^{\rho}$ in the simulation and s is the seed of the node in the GSD graph corresponding to the root of the tree in the commit, then $\mathcal{A}$ is able to distinguish $H_{\mathrm{dep}}(s)$ from $r \leftarrow \{0,1\}^{\rho}$. For $\mathcal{A}'$ to make use of this, we need to make sure that this node remains safe in the GSD graph.

More concretely, let us go over how the various queries in the PC-CGKA game can be simulated. We will refer to nodes in the GSD graph as *GSD nodes* and nodes in the TreeKEM tree as *tree nodes*. We can also model the init keys with GSD nodes, as only seeds of nodes are ever encrypted with them. $\mathcal{A}'$ can always keep track of the public state of the tree (as viewed by any user) using the reveal oracle in the SD-GSD game. For the initial create-group query or any create-commit query with a single node in the group, it suffices to create a GSD node for the leaf tree node and sample the group key of the commit u.a.r. If $\mathcal{A}$ asks to be challenged on such a commit, then we cannot make use of $\mathcal{A}$'s output in the GSD game. However, note that if such a commit is safe, then $\mathcal{A}$ is never leaked any information about the group key and has zero advantage in this case. Proposals can also be simulated easily as creating them only requires knowing public values. The leaf key pair sampled in an add proposal is of course modelled with a GSD node. To simulate the creation of a commit and corresponding welcome messages:

- $\mathcal{A}'$ can apply the proposals as in $\Sigma_{\mathrm{TK}}.\mathrm{CreateCommmit}$, since this only requires knowing public values
- use seed dependencies in the SD-GSD game to model the new nodes on the direct path
- compute the ciphertexts for the commit and welcome messsages using encryption queries in the SD-GSD game

To simulate deliver-commit and deliver-welcome, $\mathcal{A}'$ updates the public state of the target user's tree accordingly. Queries to corrupt are a bit more involved. Since $\mathcal{A}'$ can only keep track of the public state of each user, it must be prepared to compute the real group state of a user upon receiving such a query. Note however that the secret keys known by a group member can always be computed as a function of their current secret key, which can be learned using a corrupt query in the SD-GSD game, and the transcript of commits applied by the member with this secret key.

Finally, it follows from Defintion 28 that when $\mathcal{A}$ challenges a safe commit (in a group with more than one user), the corresponding GSD node that $\mathcal{A}'$ challenges is also safe.

For a detailed proof we refer the reader to [3, Theorem 12].

Theorems 2 and 3 together implying the following final result on the security of TreeKEM when DHIES is used as the public-key encryption scheme. This result was already stated informally in Theorem 1.

Theorem 4. *Let $\Sigma_{\mathrm{TK}}^{\mathrm{DH}}$ denote TreeKEM protocol instantiated with Π_{DH} (DHIES) as the public-key encryption scheme. Let Π_s the private-key encryption scheme, $\mathcal{G}$ the group-generation algorithm and H_{DH} the key-derivation function used in Π_{DH}. If Π_s is $(t, \varepsilon_{\mathrm{EAV}})$-EAV-secure and the DDH problem is $(t, \varepsilon_{\mathrm{DDH}})$-hard relative to $\mathcal{G}$ and the functions $H_{\mathrm{DH}}, H_{\mathrm{gen}}$ and H_{dep} are modelled as random oracles, then for all c, p, u with $u \geq 3$, $\Sigma_{\mathrm{TK}}^{\mathrm{DH}}$ is $(\tilde{t}, \tilde{\varepsilon}, c, p, u)$-PC-CGKA-secure with $\tilde{t} \approx t$ and*

$$\begin{aligned}\tilde{\varepsilon} &= 2 \cdot \delta \cdot N \cdot \varepsilon_{\mathrm{EAV}} + 2 \cdot N \cdot \varepsilon_{\mathrm{DDH}} + \frac{2 \cdot m_{\mathrm{DH}} \cdot N^2}{q} + \frac{m_{\mathrm{s}} \cdot N}{2^{\rho-1}} \\ \delta &:= u \\ N &:= 2 \cdot c \cdot \log(u) + u + p\end{aligned}$$

where m_{s} is an upper bound on the number of queries made to either H_{gen} or H_{dep}, m_{DH} is an upper bound on the number of queries made to H_{DH}, q is a lower bound on the size of the group output by $\mathcal{G}$ and ρ is the number of random bits sampled by $\Pi_{\mathrm{DH}}.\mathrm{Gen}$.

Proof. Combine Theorem 2 and Theorem 3, and note that $\lceil \log(x) \rceil + 1 \leq 2 \cdot \log(x)$ for $x \geq 3$