Security Analysis of the MLS Key Distribution
Chris Brzuska, Eric Cornelissen, Konrad Kohbrok
Aalto University, Finland

Abstract—Cryptographic communication protocols provide confidentiality, integrity and authentication properties for end-to-end communication under strong corruption attacks, including, notably, post-compromise security (PCS). Most protocols are designed for one-to-one communication. Protocols for group communication are less common, less efficient, and tend to provide weaker security guarantees. This is because group communication poses unique challenges, such as coordinated key updates, changes to group membership and complex post-compromise recovery procedures.

We need to tackle this complex challenge as a community. Thus, the Internet Engineering Task Force (IETF) has created a working group with the goal of developing a sound standard for a continuous asynchronous key-exchange protocol for dynamic groups that is secure and remains efficient for large group sizes. The current version of the Messaging Layer Security (MLS) security protocol is in a feature freeze, i.e., no changes are made in order to provide a stable basis for cryptographic analysis. The key schedule and TreeKEM design are of particular concern since they are crucial to distribute and combine several keys to achieve PCS.

In this work, we study the MLS continuous group key distribution (CGKD) which comprises the MLS key schedule, TreeKEM and their composition, as specified in Draft 11 of the MLS RFC, while abstracting away signatures, message flow and authentication guarantees. We establish the uniqueness and key indistinguishability properties of the MLS CGKD as computational security properties.

I. INTRODUCTION

We expect modern-day messaging applications to provide end-to-end security, guaranteeing confidentiality and authenticity of the transmitted messages. This expectation has been made a reality over the course of 25 years through the design of cryptographic protocols, which nowadays guarantee secure communication, potentially even after a participant’s key is compromised.

Our everyday communication protocols inherit their security from the properties of the keys they use, which, in turn, are established via key-exchange protocols.

Key exchange protocols therefore have been designed to achieve strong security guarantees: Entity Authentication [7], [27] ensures that the key is shared only with the intended recipient. Key Indistinguishability [20], [27], in turn, guarantees that no one but the participants involved has information about the key. A stronger variant of key indistinguishability called Forward Secrecy (FS) ensures that short-term communication keys exchanged in past sessions are secure if a party’s current state or long-term key gets compromised [27].

With the introduction of continuous key-exchange, for example by protocols such as OTR [11] or ZRTP [31], parties are able to recover from compromise if the adversary remains passive for a brief period of time. This counterpart to FS was first described as Post-Compromise Security (PCS) by Cohn-Gordon et al. [15], [22].

Modern messaging protocols require that messages can be sent even if a recipient is offline, requiring an asynchronous protocol. As a consequence, protocol sessions don’t naturally end when one participant goes offline, leading to sessions that remain active for months and years. To cover this use-case, non-interactive key exchange protocols were introduced that perform key-exchange and entity authentication continuously even in an asynchronous setting. Most notably, the Signal protocol [26] (formerly TextSecure) achieves strong FS and PCS guarantees [17], [15] due to the use of double ratcheting (formerly Axolotl), which is based on the Asynchronous Ratcheting Tree (ART) by Cohn-Gordon et al. [16].

As users of messaging protocols tend to use more than one device and/or communicate with more than one party at a time, messaging protocols are commonly required to be group messaging protocols. This gives rise to the definition of continuous group key exchange protocols by Alwen et al. [2].

MLS. The Messaging Layer Security (MLS) Working Group of the Internet Engineering Task Force (IETF) aims to create a new continuous, asynchronous group key-exchange protocol [5] that is efficient even for large groups [29]. To achieve this, the current draft makes use of the TreeKEM protocol (based on [9]).

In short, TreeKEM, which is inspired by ART, is a continuous group key distribution based on a tree structure where each leaf node represents a member’s Key Encapsulation Method (KEM) key pair and all other nodes represent a secret value and KEM key pair shared by the members in the node’s subtree. As a result, secrets can be shared efficiently by encrypting it to individual subtrees. The secret value at the root node of the tree is a secret shared by the entire group.

Each member can update their keying material by updating the key pairs and values of their leaf-to-root path, and transfer the new key material to all other members efficiently by encrypting it to the set of subtrees below the path nodes. Each member can also remove a member or add a new member, and update the key material so that previous group members cannot access key material any longer. In Section III we explain TreeKEM in more detail. The updating of keys allows members to achieve PCS and FS similar to ART.

Note that TreeKEM does not have an internal mechanism for members to reconcile differing views of individual groups and thus relies on members agreeing on a total order of group operations. One straightforward way of achieving this is a central distribution service enforcing that order.

Once a new secret value is distributed via TreeKEM, MLS combines the new secret with existing key material used for
communication. From these two keys, a new communication key is derived for the next epoch. In addition, key derivation includes information about the group and, optionally, allows to inject one or more external keys. The combination of keys allows MLS to achieve PCS and FS guarantees, and the inclusion of group parameters such as membership ensures agreement on said parameters. The processing and combining of the key material and context information is described in the MLS key schedule.

Related Work. The MLS design process has been accompanied by security analyses and proposals for changes to the protocol which we summarize below.

Bhargavan, Beurdouche and Naldurg (BBN, [10]) use $F^*$ to create a novel symbolic verification tool and use it to perform a symbolic analysis of MLS Draft 7. They uncover two attacks and several other weaknesses in the protocol that are subsequently fixed.

In [2], Alwen, Coretti, Dodis and Tselekounis (ACDT) introduce the Continuous Group Key Agreement (CGKA) security notion and introduce the notion of Post-Compromise Forward Security (PCFS), which encodes the security of a given session despite state compromise both before and after the session. They analyze the security of TreeKEM in MLS Draft 7, finding that while it provides the desired Post-Compromise Security (PCS) guarantees, its Forward Secrecy (FS) guarantees are very weak. They propose to modify TreeKEM in order to improve FS guarantees based on an algebraic KEM construction that requires keys to be updatable.

Alwen, Capratto, Cueto, Kamath, Klein, Markov, Pascual-Perez, Pietrzak, Walter and Yeo (ACCK+ [1]) build on ACDT by proving the CGKA security of TreeKEM in MLS Draft 9 in a stronger adversarial model. They also propose the notion of Tainted TreeKEM as an alternative to TreeKEM with a different performance profile, but with comparable security.

Following ACCK+, Alwen, Coretti, Jost and Mularczyk (ACJ, [3]) prove the CGKA security of TreeKEM in MLS Draft 9 against an even stronger adversary, allowing active interference of the adversary in the protocol flow.

Continuing the work of ACJ, Alwen, Jost and Mularczyk (AJM, [4]) analyze the CGKA security of TreeKEM in MLS Draft 10 with regard to insider security, formalizing and proving the as-of-yet strongest security notion for TreeKEM, with their work resulting in a number of changes to the MLS protocol. Even though in this work we only analyze a subset of MLS (compared to the whole protocol as analyzed by AJM) and with a different adversarial model, we reach a similar conclusion in terms of the security of MLS. See Section [V] for a more detailed comparison of model and results.

Independent of other works, Weidner [30] proposes an alternative version of TreeKEM, named Causal TreeKEM, that does not require a strict total order of groupoperations. This would allow a relaxation of MLS' requirement of a strict order of handshake messages, but similar to the proposal by ACDT would require a specialized KEM construction.

In the only work considering the multi-group setting, Cremers, Hale and Kohbrok (CHK, [19]) compare MLS with a straightforward group messaging protocol made of 1:1 Sig-
Figure 1: The tree represents a group with members A, B, C, D, E, G and H. Inside a node, we display TreeKEM secrets known to members in the corresponding sub-tree. The pink values are used in the derivation and not stored long-term. The blue part is only used for external add operations.

II. PRELIMINARIES AND NOTATION

Binary Trees. A binary tree is a finite, connected, and directed graph with a single source — called root — such that each node has at most two outgoing edges. For two nodes \( n_0 \) and \( n_1 \), if there is an edge \( n_0 \rightarrow n_1 \), then we call \( n_0 \) the parent of \( n_1 \) and \( n_1 \) the child of \( n_0 \). All nodes which can be reached from \( n_0 \) are its descendants, and all nodes from which \( n_1 \) can be reached are its ancestors. Every node can be reached starting from the root, and we call the set of ancestors of a leaf node its direct path. We allow for nodes to be marked as blank (the marking will later take on the meaning of no associated data). We call the co-path a list of the closest non-blank descendents of the nodes along the direct path.

We leave the encoding of the tree abstract in this paper and only assume that given the size of a tree and the index \( i \) of a node, we can find the indices of its direct path.

Notation. We use pseudocode to describe algorithms. We denote sets and tables by capital letters, e.g. \( S \) or \( T \). We denote algorithms in lowercase and sans serif. \( x \leftrightarrow a \) assigns value \( a \) to variable \( x \), and \( x \leftarrow \text{algo}(a) \) runs algo on value \( a \) and assigns the result to variable \( x \). When algo is a randomized algorithm, \( x \leftarrow \text{algo}(a) \) runs algo on value \( a \) with fresh randomness and assigns the result to variable \( x \). When \( S \) is a set, \( x \leftarrow S \) samples a uniformly random value from \( S \) and assigns that value to variable \( x \). A common set is \( \{0, 1\}^\lambda \), the set of bitstrings of length \( \lambda \). We use \( \pi \) for a vector of variables, and \( \pi \overset{\text{cfgkd}}{\leftarrow} \text{algo}(\pi) \) means, we run algo separately on each value in the vector \( \pi \) and then assign the results to the corresponding indices in vector \( \pi \). Analogously, \( \pi' \overset{\text{cfgkd}}{\leftarrow} S \) means that we sample each value in the vector \( \pi \) independently and uniformly at random. For two tables \( T \) and \( U \), the operation \( T \overset{\text{merge}}{=} U \) overwrites values \( T[i] \) by values \( U[i] \) whenever \( U[i] \) is defined, and the operation \( T \overset{\text{rem}}{=} U \) removes all values \( T[i] \) from \( T \) where \( U[i] \) is defined. assert \( \text{cond} \) abbreviates that a special error message is sent unless \( \text{cond} \) holds.

III. MLS CONTINUOUS GROUP KEY DISTRIBUTION

A continuous group key distribution protocol (CGKD) allows clients to form a group. A client that is in a group is called a member. A group allows members to maintain a continuous session which

- distributes a secret value among group members in each epoch;
- allows for efficient updates to the key material of a member and the group;
- allows for efficiently adding clients as new members to the group;
- allows for efficiently removing members from the group.

Assuming pseudorandomness and uniqueness of input keys, a CGKD achieves pseudorandomness and uniqueness of output keys. It does not model signatures and attacks against authentication, but instead allows the adversary to combine keys (almost) arbitrarily and provide guarantees whenever at least one of the base keys is honest—see Section VI.
for how the security guarantees of the CGKD relate to the security properties of a protocol which uses the CGKD which seeks to achieve forward-secrecy, post-compromise security and security against insider attacks. In Appendix A, we define the syntax of a CGKD formally, see Figure 2 for the inputs and outputs of the algorithms process, process and join. In the body, we only describe the MLS CGKD informally, since we provide security models which are specialized to the MLS CGKD. We split the MLS CGKD into a weak variant of continuous group key distribution (wCGKD) and the syntax of a key schedule (KS). The main difference between a CGKD and a wCGKD (often referred to as TreeKEM) is that the group keys of a wCGKD are corrupt whenever one group member’s current private keys are corrupted. This property is, indeed, rather weak, since in large groups, individual members might use weak randomness, and for insecurity, it suffices if a single member of the group relies on weak randomness for generating their own keys. In turn, a (strong) CGKD relies on key material which is chained across epochs. If the key material from a previous epoch was secure, then security in the current epoch is maintained—unless the key material is leaked by encrypting it to a client with corrupt keys that joins the group. As we will see for MLS, a wCGKD can be combined with a key schedule to obtain a CGKD.

Updates of the key material are initiated by one group member and we refer to this procedure as (w)update in (w)CGKD. In order for other group members to keep their key material in sync with an updating party, they need to process the messages generated by the update algorithm. Since new group members might perform processing in different way than existing group members, the join algorithm models the computation performed by new joiners. Last, but not least, pke.kgen allows to generate new public keys. We now describe the MLS CGKD, MLS wCGKD and MLS KS.

**MLS Key Schedule.** The purpose of the key schedule is to combine existing key material into new key material and, in particular, to chain keys across epochs via the internal key \( s_{\text{int}} \), which is referred to as the *init secret* in MLS terminology. In addition, the MLS key schedule allows to combine \( s_{\text{int}} \) with an external key \( s_{\text{ext}} \), PSK in the MLS RFC, and a commit secret \( k_{\text{cs}} \), which is derived via the MLS wCGKD. We abstract away the details of how MLS pre-processes the external key material in case that several external keys are combined and work directly with a single \( s_{\text{ext}} \).

The MLS key schedule \( k_s \) uses a key derivation function (KDF) which is constructed based on an extract (XTR) and an expand (XPD) function, following Krawczyk’s HKDF standard [24]. \( k_s \) chains two full KDFs and then a single XPD function, see top of Figure 1. In OPTLS [25], it was suggested to use the salt position in HKDF to combine two keys, and therefore, the chained call of two KDFs allows to combine the aforementioned three secrets \( s_{\text{int}}, s_{\text{ext}}, \) and \( k_{\text{cs}} \). See Figure 1 for an illustration of the MLS Key Schedule algorithm \( k_s \). Note that the join algorithm of the MLS CGKD only knows \( k_{\text{ps}} \), but not \( k_{\text{cs}} \) and thus starts the computation of the key schedule only at \( k_{\text{ps}} \). The key schedule derives several secrets and, in particular, uses one of the secrets as input to a *key pair* derivation function (DKP), which is a deterministic algorithm that corresponds to running pke.kgen using its input as explicit randomness. We now turn to the MLS wCGKD.

**MLS wCGKD.** The MLS wCGKD is called TreeKEM, and we use the terms MLS wCGKD and TreeKEM interchangeably in this work. The main purpose of TreeKEM is to provide new \( k_{\text{cs}} \) values to ks. I.e., \( k_{\text{cs}} \) is fresh key material which is shared between all group members. As the name suggests, key material in TreeKEM is structured and stored in a tree data structure. The leaf nodes of the tree represent group members, the other nodes represent shared secrets known by members in the subtree, i.e., members which correspond to the leaves in the subtree below a node. E.g., \( k_{\text{cs}} \) is shared by all members of the group and indeed derived from the value \( ps \) which is stored at the root of the tree. Figure 3 illustrates the data associated with a group key distribution (w)CGKD. In OPTLS [25], it was suggested to use the salt position in HKDF to combine two keys, and therefore, the chained call of two KDFs allows to combine the aforementioned three secrets \( s_{\text{int}}, s_{\text{ext}}, \) and \( k_{\text{cs}} \). See Figure 1 for an illustration of the MLS Key Schedule algorithm \( k_s \). Note that the join algorithm of the MLS CGKD only knows \( k_{\text{ps}} \), but not \( k_{\text{cs}} \) and thus starts the computation of the key schedule only at \( k_{\text{ps}} \). The key schedule derives several secrets and, in particular, uses one of the secrets as input to a *key pair* derivation function (DKP), which is a deterministic algorithm that corresponds to running pke.kgen using its input as explicit randomness. We now turn to the MLS wCGKD.

**MLS wCGKD.** The MLS wCGKD is called TreeKEM, and we use the terms MLS wCGKD and TreeKEM interchangeably in this work. The main purpose of TreeKEM is to provide new \( k_{\text{cs}} \) values to ks. I.e., \( k_{\text{cs}} \) is fresh key material which is shared between all group members. As the name suggests, key material in TreeKEM is structured and stored in a tree data structure. The leaf nodes of the tree represent group members, the other nodes represent shared secrets known by members in the subtree, i.e., members which correspond to the leaves in the subtree below a node. E.g., \( k_{\text{cs}} \) is shared by all members of the group and indeed derived from the value \( ps \) which is stored at the root of the tree. Figure 1 illustrates the data associated with a group key distribution (w)CGKD. In OPTLS [25], it was suggested to use the salt position in HKDF to combine two keys, and therefore, the chained call of two KDFs allows to combine the aforementioned three secrets \( s_{\text{int}}, s_{\text{ext}}, \) and \( k_{\text{cs}} \).

Figure 3: (a-c): Member A updates their their key material. (a) original tree for the group. (b) tree during the update, highlighting the direct path in black and the co-path in dark grey. (c) tree during the update if there are blank nodes in the tree. (d-f) Adding a new member to a group. (d) tree for the original group. (e) tree with \( H \) added to the group in an add-only operation, highlighting blanked nodes in white. (f) tree for the update performed by A, highlighting the direct path in black and the co-path in dark grey. Note that H’s unmerged leaf is part of the co-path. (g-i) Removing a member from a group. (g) tree for the original group. (h) tree with \( H \) removed. (i) tree for the update after the removal, highlighting the direct path in black and the co-path in dark grey. Blanked nodes are highlighted in white in both (h) and (i).
with each node as well as the members to which the secret node data is known. Every node in the tree has associated with it a KEM key pair known to the members represented by the leaves in the sub-tree. To share a secret value with some members, a member can encrypt the value either to the public key of a node whose secret key all intended members know or encrypt to several public keys for a more fine-grained information-sharing.

The main sub-procedure of the TreeKEM is the Derive Node procedure $dn$, see Figure [1]. $dn$ uses an expand function (XPD) and a key encapsulation mechanism (KEM) as building blocks. $dn$ takes as input the secret of a node, referred to as the path secret $ps_i$ of node $i$, and uses XPD to derive two random values: The path secret $ps_{p(i)}$ (where $p(i)$ refers to the parent of node $i$) and randomness for (DKP) to computes a KEM key pair $(pk_i, sk_i)$ associated with node $i$.

Roughly, in the tree of Figure [1] each arrow can be interpreted as a possible application of $dn$ with the restriction that if a node $i$ has two non-blank children, then its path secret was only derived from one of them. The keys associated with each node evolve over time via updates which start from a leaf node and proceed along the direct path to the root. The path secret $ps_i$ of a node $i$ is derived from the child which was last contained in the direct path of an update.

We next explain key updates. In addition, one can also combine an update with the adding and/or the removing of one or more members. While adding and removing is implemented simultaneously, we describe the conceptual idea for each separately below.

**Updating Key Material.** Consider the group represented by the tree in Figure [3a]. The group member Alice ($A$) can update the group secret by the following process. First, $A$ samples a random value $k_{upd}$ for the key update and uses it as the new path secret associated with $A$’s leaf node, i.e., $ps ← k_{upd}$. Then, Alice runs $(ps', pk, sk) ← dn(ps)$ and stores $pk$ in $PK'_{mem}[i_{cur}]$ and $PK'_{upd}[i_{cur}]$, where $i_{cur}$ is the index of Alice’s leaf node, $PK'_{mem}$ is a table which represents Alice’s view of the public keys of the group members, and $PK'_{upd}$ is a table which contains the public-keys that Alice will tell other group members to change. Additionally, Alice also stores the secret key in $SK'_{aum}[i_{cur}]$. $SK$ is a table of secret keys which Alice knows. Then, Alice computes the parent index of $i_{cur}$ and proceeds in the same way for the parent index. First, she computes $(ps'', pk, sk) ← dn(ps')$, then stores $pk$ and $sk$ in the adequate tables, then moves to the parent node and applies $dn$ to $ps''$ etc. until reaching the root node. This loop is captured by $dopath$ which is hinted at in Figure [1] Note that $dopath$ needs to encrypt to the entire co-path and not only to a single node as hinted at in Figure [1]. Jumping ahead to the pseudocode provided in Figure [17] encryption to the co-path is only executed if $k_{upd}$ is non-empty. Note that the assert in line 2 of the wupdate code only forces $k_{upd}$ to be defined if a member is removed or if an external add is performed (since in both of these cases, the keys needs to be updated). In the end, Alice has replaced all the path secrets, public-keys and secret-keys of the direct path, marked in black in Figure [3b].

As only Alice knows the path secrets used along this path, she now also shares the path secret with the other group members. Thus, for each node $i_{cur}$ on her direct path, she considers the child $i_{child}$ of $i_{cur}$ which is not on her direct path (but, instead, is on the co-path) and encrypts the path secret of $i_{cur}$ under $PK_{mem}[i_{child}]$, the public-key associated with this node. In this way, the entire sub-tree below $PK_{mem}[i_{child}]$ can decrypt the ciphertext. In Figure [3b] the co-path is marked in grey, i.e., for each grey node, Alice encrypts to the public-key associated with it.

Figure [3a] and Figure [3b] now illustrate a case where there are no blank nodes in the tree—blank nodes are nodes which do not have data associated with it. Figure [3c] depicts the case that the tree contains some blank nodes. In this case, Alice encrypts to the nodes marked in grey in Figure [3c]. Note that we nevertheless refer to these nodes as co-path even though our co-path definition does not coincide with the graph-theoretic notion of co-path. In order to simplify the computation of the co-path, we give a description of the co-path $CP$ as input to update. $CP$ is a table of sets, i.e., for each tree index $i$, $CP[i]$ contains a set of the indices to which Alice needs to encrypt. It is a set because Alice may need to encrypt to multiple public keys due to blank and unmerged nodes, see Figure [17] for details.

**Adding a New Member.** There are two ways to add a new member. One can either add a member in an add-only operation, or one can add a member and perform an update at the same time. (In this case, one can also remove members simultaneously, but let us ignore this operation for now.) When performing an add operation, we either find the left-most empty leaf node (there might be empty leaf nodes due to previous removals) or, if the tree is full, we replace a leaf node by a 3-node sub-tree. E.g., consider the original group in Figure [3d]. Here, to add Henry ($H$) to this group, the group member Alice ($A$) blanks Bob’s ($B$) node, creates below it a node for $H$ and a new node for $B$ and associates $H$’s node with a KEM public-key which Alice knows belong to $H$, and associates $B$’s new node with the same public-key which $B$ had before. Moreover, Alice runs the key schedule with an empty $k_{upd}$ and encrypts $k_{join}$ to $H$.

In the case that $A$ wants to simultaneously perform an update of the key material, she now updates her path as described in Section [3] and encrypts the relevant path-secrets to the co-path, marked in grey. Again, the co-path is not the graph-theoretic co-path, since $A$ needs to encrypt a message to $H$ separately. In addition to sharing the relevant path secrets with the co-path, $A$ also encrypts the joiner secret $k_{join}$ via the MLS Key Schedule (see Figure [11]) to $H$. If Alice performs an add-only (without an update), she performs the computations in the same way except that $k_{ext}$ is now a fixed zero value. It is important to execute the ks function also in an add-only operation to ensure that the new joiners cannot read messages from the time before they joined.

A special case is the situation where the add operation is not performed from inside the group, but instead, an outside member adds itself to the group. In this case, the external group public-key $pk_{ext}$ is used (marked in blue in Figure [1]
and Figure 17 to transmit s_int. Note that the pink code in Figure 17 is only run if \( pk_{\text{ext}} \) is non-empty, which requires that \( k_{\text{upd}} \) is non-empty, too.

Removing a Member. Consider the group represented by the tree in Figure 29. To remove a group member Henry (H) from this group, the group member Alice (A) blanks the nodes on the path from node H to the root, shown in Figure 31. As a result, a co-path that included those nodes before, now, instead includes their non-blank child nodes.

Notice that the removal by itself does not update the group secret and therefore, A also needs to perform an update to ensure that H cannot decrypt messages sent by group members anymore. The requirement to perform an update is captured by

if \[ |PK_{\text{rem}}| > 0 \] : assert \( k_{\text{upd}} \neq \perp \)

in line 2 of the update procedure in Figure 17. As the tree now contains blanked nodes, the path secrets along the new direct path must be encrypted under the keys of a larger co-path, as shown in Figure 31.

MLS CGKD. Combining TreeKEM and ks as depicted in Figure 1 yields the MLS CGKD. As mentioned in the beginning of the section, we can now see that the MLS CGKD indeed achieves significantly stronger security properties than the wCGKD. Namely, key secrecy is violated if not yet (if any) of the group keys are compromised (and not yet updated). In turn, the MLS CGKD also yields secrecy if the \( s_{\text{ext}} \) or the \( s_{\text{int}} \) from a previous epoch are fresh. See Section VI for a discussion of key freshness.

IV. Assumptions

Game-Based Security Notions. We formulate security properties via indistinguishability between games, a real game which models the real behaviour of a cryptographic primitive, and an ideal game which models an ideally secure variant of the primitive. For example, security of a pseudorandom function is captured by demanding that the input-output behaviour of the (keyed) pseudorandom function is indistinguishable from the input-output behavior of a truly random function which draws a uniformly random output for each input.

Especially since Bellare and Rogaway promoted code-based game-hopping [8], it has been popular to formulate games in pseudocode, and we follow this approach here. I.e., a game \( G \) provides a set of oracles to the adversary \( A \), these oracles are specified by pseudocode and operate on a common state which is secret to the adversary. This hidden state might contain, e.g., a secret key. In this work, we denote by \( A \circ G \) that the adversary \( A \) interacts with the oracles of game \( G \). I.e., the adversary is the main procedure that makes queries to the oracles of \( G \), which returns an answer to \( A \), and then, in the end, the adversary \( A \) returns a bit indicating their guess whether the game is indeed real or ideal. The advantage \( \text{Adv}_{G^0,G^1}(A) \) measures an adversary \( A \)’s ability to distinguish between \( G^0 \) (real) and \( G^1 \) (ideal).

**Definition IV.1 (Advantage).** For an adversary \( A \) and two games \( G^0,G^1 \), the advantage \( \text{Adv}_{G^0,G^1}(A) \) denotes the term
\[
\left| \Pr \left[ 1 = A \circ G^0 \right] - \Pr \left[ 1 = A \circ G^1 \right] \right| .
\]

Figure 4: Definition of the KEY and LOG package, where \( \text{reg}(\cdot) \) is an injective handle constructor, creating a tuple from the inputs. Oracle REG is useful since it can be exposed to the adversary without leading to malformed handles, unlike SET which allows the caller to specify arbitrary handles. The level function returns whether the handle is from a SET or REG query.

We formulate security statements by relating advantages. E.g., we upper bound an adversary \( A \)’s advantage against a game pair for MLS key distribution security by an adversary \( B \)’s advantage against a game pair for the base primitive (e.g., a pseudorandom function) security. Importantly, the runtime of \( A \) and \( B \) will be similar. Namely, we often write \( B \) as \( A \circ R \) where \( R \) is referred to as a reduction which translates \( A \)’s oracle queries to its own game (in this case, the game pair for MLS key distribution security) into oracle queries to the game for the base primitive.

State-Separating Proofs. The previous description lacks a natural way to compose games. However, if games are not described as a single block of pseudocode but rather sliced into individual packages of code, then the same package of code can be reused. This notion of code-reuse, together with a separate state for each package results in a natural notion of package composition to form games.

A frequent example of a natural, re-usable code package is the KEY package described in Figure 4. Cryptographic
primitives often share key material (e.g., in the MLS key schedule, KDF produces outputs which become keys of KDF which produces outputs which become keys of XPD etc., see Figure 1). Now, if KDF computes keys and stores them in a KEY package using the SET oracle, then XPD can retrieve them via the GET oracle.

We now explain how to use KEY packages to model security. For convenience we use orange boxes to indicate \( b = 0 \) and blue boxes to indicate \( b = 1 \). Figure 6 and Figure 7 describe the security games for KDF, XPD and HPKE, decomposed into several packages. For KDF and XPD, the upper KEY models the input key, the lower KEY packages the output keys. KDF and XPD are stateless packages, (see Figure 6d) which retrieve an input key from the upper KEY package, perform a computation, and store the result in the lower KEY package. The lower KEY package models security, namely pseudorandomness of the output keys.

Recall that our security notion aims to capture that output keys are indistinguishable from uniformly random values of the same length. Thus, if \( b = 0 \) in KEY\( ^{h,\lambda} \), the concrete key values are stored, and if \( b = 1 \) in KEY\( ^{h,\lambda} \), then uniformly random keys of the same length are drawn. Finally, the adversary can retrieve the output keys via the GET oracle.

Note, that we want to model multi-instance security, where some of the keys might be known to the adversary. We track the adversary’s knowledge of a key by marking the key as dishonest (sometimes also called corrupt). In that case we will say that the honest value \( hon = 0 \). In turn, for honest keys, \( hon = 1 \). In the case that \( hon = 0 \), the concrete keys (i.e., the real result of the key computations) are stored regardless of whether \( b = 1 \) or \( b = 0 \). It is important that the adversary can access also honest keys via GET and not only dishonest keys, since the game models pseudorandomness of honest keys, and for dishonest keys, the ideal and real game behave identically.

We model multi-instance security with static corruptions on the input keys and corrupt-upon-derivation (CUD) on the output keys. If all input keys are corrupt, then the output value is marked as corrupt as well. Moreover, if the adversary uses \( hon = 0 \) as a parameter in its oracle query to EVAL, then the key will be marked as corrupt, too. Else, keys are marked as honest. Note, that in the upper KEY package, honest keys are sampled uniformly at random and secret from the adversary, as the adversary cannot access them via GET, because the GET query cannot be asked by the adversary—the arrows specify which package may call which oracles of which package. Importantly, this means that the graphs are part of the formal game definitions.

**Handles.** If an adversary wants to interact with a specific key, e.g., to retrieve it via a GET or have it be the subject of another oracle query such as EVAL, they use the handle of that key. We construct handles in such a way that each handle injectively maps to a key. When a key is initially randomly generated via the REG oracle, its handle \( \text{reg}() \) is composed of the index of the package KEY and a counter, implemented as \( [K] \) where \( K \) is the table of keys already stored in the KEY package, making it globally unique. We denote by \( \text{reg}() \) the corresponding and injective handle constructor, see Figure 4.

If a key is derived from one or more other keys, the handle of the output key is constructed from the handles of the input key(s), as well as any other inputs to the key derivation. See, e.g., the description of the EVAL oracles in Figure 6e. This guarantees that the handle-to-key mapping is computationally injective, i.e., will not have collisions unless we found a collision in the key derivation function used to derive the key.

Finally, let us elaborate on the role of the KEY package in HPKE. Here, KEY models a message that shall remain secret. If the HPKE is idealized, the message is instead replaced with an all-zero string before encryption (Figure 7). Note that pairs of secret keys and public keys are stored in the KEY package, as Figure 5 and Figure 7.

**Logging.** We now turn to how we model collision resistance and key uniqueness. We call two handles \( h \) and \( h' \) such that they correspond to the same key in the \( K \) table in KEY a key collision. For honest, uniformly random keys, such collisions are unlikely, but for keys registered by the adversary or known to the adversary, such collisions can trivially occur. We thus 1) disallow the adversary from registering the same dishonest key value \( k \) twice under two different handles (since identical input keys would lead to identical output keys), 2) remove collisions between registered and set dishonest keys (once) and 3) prove that then derived keys do not have collisions. We model both properties using a LOG package which keeps track of the keys and handles used so far and sends a special abort message when a collision occurs.

This modular style of code-writing has recently been developed by Brzuska, Delignat-Lavaud, Fournet, Kohbrok and Kohlweiss (BDFKK 13) and been coined state-separating proofs (SSP) since the state separation of the games enables proofs which rely on code-separation and state-separation. We now state all our assumptions formally using the composed games style and return to proofs in Section VIII-A.

**Expand.** We consider Krawczyk’s XPD function as a function \( \text{xpd} : ([0,1]^{\lambda} \times [0,1]^{\lambda}) \rightarrow [0,1]^{\lambda'} \), where \( \lambda' = \lambda(\lambda) \) is hardcoded (rather than provided as an explicit input). The second input is a pair \((\text{lbl}, \text{ctx})\), where in some cases, \( \text{ctx} \) might be empty. We parameterize our security game for XPD...
with a list of labels $\overline{lb}$ and measure the security of the XPD function by the advantage which adversaries $A$ have in distinguishing the real game $\text{GEXP}^{\overline{lb}}_{\text{xpd}}$ and the ideal game $\text{GEXP}^{\overline{lb}}_{\text{xpd}}$, defined in Figure 6a. For an adversary $A$, we define the advantage function $\text{Adv}^{\overline{lb}}_{\text{EXP}}(A) := \Pr [1 \leftarrow A \circ \text{GEXP}^{\overline{lb}}_{\text{xpd}}] - \Pr [1 \leftarrow A \circ \text{GEXP}^{\overline{lb}}_{\text{xpd}}].$

In asymptotic terminology, if the advantage $\text{Adv}^{\overline{lb}}_{\text{EXP}}(A)$ is negligible for all PPT adversaries, then xpd is a secure pseudorandom function. $\text{Adv}^{\overline{lb}}_{\text{EXP}}(A)$ incorporates both, pseudorandomness and collision-resistance into a single assumption, since the bit in the $\text{LOG}$ also changes from 0 to 1. While this is convenient in proofs, in some cases, we need pseudorandomness only, since collision resistance has already been taken care of. For an adversary $A$, we write $\text{Adv}^{\overline{lb}}_{\text{PREXP}}(A) := \Pr [1 \leftarrow A \circ \text{GEXP}^{\overline{lb}}_{\text{xpd}}] - \Pr [1 \leftarrow A \circ \text{GEXP}^{\overline{lb}}_{\text{xpd}}].$

**KDF.** We model Krawczyk’s HKDF as a function $\text{kdf} : \{0, 1\}^{\lambda} \times \{0, 1\}^{\lambda} \times \{0, 1\}^{e} \rightarrow \{0, 1\}^{\lambda’}$ for some hard-coded $\lambda’ = \lambda'(\lambda)$ such that a triple $(k, \overline{lb}, \overline{ctx})$ is mapped to a string of length $\lambda’$. We assume that $\text{kdf}$ is a dual KDF. I.e., if either of the inputs is random and secret from the adversary, then $\text{kdf}$ produces a pseudorandom output. Idealization based on the left input is modeled by a bit in the code of $\text{KDF}$. For an adversary $A$, we define the advantage $\text{Adv}^{\overline{lb}}_{\text{KDF}}(A) := \Pr [1 \leftarrow A \circ \text{GKDF}^{\overline{lb}}_{\text{kdf}}] - \Pr [1 \leftarrow A \circ \text{GKDF}^{\overline{lb}}_{\text{kdf}}],$

where $\text{GKDF}^{\overline{lb}}_{\text{kdf}}$ is defined in Figure 6b. Note that here, right inputs might come from one of many packages. In turn, idealization based on the right input is modeled by a bit in the lower $\text{KEY}$ and for an adversary $A$, is captured by the advantage $\text{Adv}^{\overline{lb}}_{\text{KDF}}(A) := \Pr [1 \leftarrow A \circ \text{GKDF}^{\overline{lb}}_{\text{kdf}}] - \Pr [1 \leftarrow A \circ \text{GKDF}^{\overline{lb}}_{\text{kdf}}],$

where $\text{GKDF}^{\overline{lb}}_{\text{kdf}}$ is defined in Figure 6c.

**Public-Key Encryption.** Lastly, we assume the existence of a public-key encryption (PKE) scheme $\text{pke}$ which consists of three algorithms $\text{pke}.\text{gen}$, $\text{pke}.\text{enc}$ and $\text{pke}.\text{dec}$ with standard correctness and confidentiality properties, namely indistinguishability under chosen ciphertext attacks (IND-CCA2). IND-CCA2 captures that encryptions of (adversarially chosen) messages $m$ are computationally indistinguishable from encryptions of $0^{\|m\|}$, even when the adversary can use a decryption oracle (except on the challenge ciphertexts). We use the PKE scheme exclusively to model Hybrid Public Key Encryption (HPKE), i.e. we only encrypt symmetric keys with the PKE scheme. The length of the symmetric keys is $\lambda$. For an adversary $A$, we define the advantage as $\text{Adv}^{\overline{lb}}_{\text{HPKE}}(A) := \Pr [1 \leftarrow A \circ \text{GHPKE}^{\overline{lb}}_{\text{pke}}] - \Pr [1 \leftarrow A \circ \text{GHPKE}^{\overline{lb}}_{\text{pke}}].$
CGKD. A crucial component in the modeling are different keys, inputs and outputs as described in Figure [1]. Based on the rationale outlined in Section [IV] storing keys in separate packages is useful for composition, and so, our model stores each type of key in a separate KEY package, i.e., there are two main packages wCGKD (modeling TreeKEM) and KS, each reading and writing keys from/into the different KEY and PKEY packages, see Figure 8a for the composed game GCGKD and Figure 8c for GKS. Figure 8b and 8c respectively show the security definition of TreeKEM and the Key Schedule on their own. Note that each KEY package has a different subscript and so have their oracles to ensure that queries are unique across the entire graph. We now map the inputs and outputs described in Figure 1 to the KEY packages.

The values (pk, sk) of the KEMs associated with different nodes in the tree are stored in PKEY_NKP packages, the path secrets are stored in the KEY_PS packages with other path secrets (incuding $k_{xpd}$ value which the updated samples). Finally, values for the $s_{ext}$ variable are stored in KEY_PSK packages, values for the $s_{int}$ variable are stored in KEY_IS packages, and values of each of the $k$ variables in $k_{ijk}$ are stored in KEY_GR.

Our base keys are modeled as uniformly random, unique keys (for honest keys) or adversarially chosen (but still unique keys), e.g., the KEY_PSK starts with an idealization bit $b = 1$, reflecting that keys are sampled uniformly at random and that the adversary cannot register the same dishonest key twice. We can then rely on the assumptions stated in Section [IV] to prove that a game where the idealization bits of the other KEY packages are all 0 is computationally indistinguishable from a game where the idealization bits of the other KEY packages are all 1. This establishes that the MLS CGKD provides pseudorandom and unique keys.

**Definition V.1 (CGKD advantage).** For an expand function $xpd$, a KDF $kdf$, a PKE $pke$, for all polynomials $d$ and $t$ and all adversaries $A$ which generate trees of depth at most $d$ and run groups for at most $t$ epochs, we denote by $Adv^{xpd,kdf,pke}(A)$ the advantage

$$
\Pr \left[ 1 \leftarrow A \circ GCGKD_{xpd,kdf,pke}^{0,d,t} \right] - \Pr \left[ 1 \leftarrow A \circ GCGKD_{xpd,kdf,pke}^{1,d,t} \right].
$$

**wCGKD** In Figure 8b the wCGKD package represents the composition of packages shown in the center of Figure 9a (LAYER0.d and EXP). Note that the KEY, PKEY, and PKE packages correspond to those in Figure 8b. The LAYER packages represent an at most $d$-layer tree structure where each layer is encoded as a separate composition of packages. Note that for each layer, the KEY_PSK package, storing path secrets, has a REG query that allows nodes on that layer to function as a leaf node by generating a random (secret) value from scratch.

We define each LAYER package as composition of packages, see Figure 8b. Again, the KEY, PKEY_NKP, and PKE packages correspond to those in Figure 8b. Figure 8b encodes both the dn function (Figure 1) through the EXP and DKP.
packages, and the encryption of path secrets to other members through the HPKE, PKE, PKEY, and KEY_PS packages. The GET access of HPKE to all layers above it encodes that path secrets at various positions in the tree might be encrypted by HPKE.

This model gives significantly more power to the adversary than the MLS TreeKEM interface would (making our security statements only stronger). The model allows the adversary to set up derivations and encryptions that encode arbitrary tree structures. The model merely enforces the concept of layers.

**Definition V.2 (wCGKD advantage).** For an expand function $\text{xp}d$, a PKE $\text{pke}$, for all polynomials $d$ and all adversaries $A$ which generate trees of depth at most $d$, we denote by $\text{Adv}_{\text{wCGKD}}(A)$ the advantage

$$\Pr \left[ 1 \leftarrow A \circ \text{wCGKD}^{0,d}_{\text{xp}d,\text{pke}} \right] - \Pr \left[ 1 \leftarrow A \circ \text{wCGKD}^{1,d}_{\text{xp}d,\text{pke}} \right].$$

**Key Schedule model.** We define the $\text{KS}$ package in Figure 8c by the composition of packages shown in the center of Figure 10b (EPOCH$_{0..t}$). The KEY, PKEY, and PKE packages correspond to those in Figure 8c. The EPOCH packages represent a $t$-epoch group where each epoch is encoded as a separate composition of packages. Each EPOCH package is the composition of packages shown in the center of Figure 10b. Recall that the KEY, PKEY, and PKE packages correspond to those in Figure 8c.

Figure 10b encodes the key schedule $ks$ (Figure 1). The KDF and EXP packages correspond to kdf and xp calls, and the DKE package corresponds to the dkp call. The HPKE packages encode the encryption of the joiner secret to new members (top), and the encryption of init secrets by external committers (bottom). The REG query allows the adversary to create external init secrets. The values of external key pairs, $(pk_{\text{ext}}, sk_{\text{ext}})$, are stored in the PKEY_EX package.

**Definition V.3 (KS advantage).** For an expand function $\text{xp}d$, a key derivation function $\text{kdf}$, a PKE $\text{pke}$, for all polynomials $d$, $t$, and all adversaries $A$ which generate trees of depth at most $d$ and run groups for at most $t$ epochs, $\text{Adv}_{\text{KS}}^{d,t,\text{xp}d,\text{pke},\text{kdf}}(A)$ denotes

$$\Pr \left[ 1 \leftarrow A \circ \text{KS}^{0,d,t}_{\text{xp}d,\text{pke},\text{kdf}} \right] - \Pr \left[ 1 \leftarrow A \circ \text{KS}^{1,d,t}_{\text{xp}d,\text{pke},\text{kdf}} \right].$$
Each freshness condition follows the same pattern, namely, the honesty of the output key is a function of the honesty of the keys it is derived from. We can express the honesty of a key by a honesty graph, which is a directed, acyclic graph, where each edge represents the boolean honesty value of a key and each node represents an operation the values of its incoming edges. Edges without an origin or destination node represent the honesty values of input or output keys, respectively.

If a key is derived from a single key, it either inherits that key’s honesty or it is immediately corrupted by the adversary. This mechanism is visible in the definition of EVAL oracle of the EXP package (Figure 6), where the adversary can pass in an hon value to indicate if they want the output key to simply inherit the honesty of the input key, or if they want the output key to be dishonest.

If a key is derived from two keys, it is honest if one of the input keys is honest and the adversary doesn’t decide to compromise it immediately. Similar to single-key derivation, this mechanism is visible in the definition of the EVAL oracle of the GKD package (Figure 6).

Therefore, we can now express the honesty of any key in the graph as a function of the honesty of the root keys from which the key is derived. Namely, the key is honest only if there is at least one root key which is (a) honest and (b) where none of the keys on the path from the root key to is corrupted.

We provide the honesty graph which is induced by CGKD in Figure 11. The honesty graph of each individual output key is a subgraph of the entire model’s structural graph, and the honesty graphs of key schedule and wCGKD are subgraphs of the CGKD graph, see Figure 11. In addition, in each derivation, the adversary can corrupt upon derivation, and thus, the resulting honesty value of an (intermediate or output) key is always the ∧ of the honesty value shown in Figure 11 and the value provided by the adversary in the derivation. For simplicity, we do not depict the CUD in the graph—it would amount to adding a ∧ node right after each derivation.

Forward Secrecy (FS). Traditionally, key exchange models define forward secrecy in terms of long term keys and in a scenario where sessions terminate. The first does not apply,
as our model does not consider long term keys. The second does not apply, because MLS considers long-term group conversations that do not explicitly terminate. Thus, messaging protocols — and MLS in particular— seek to achieve fine-grained FS by constantly deriving keys, such that old keys can be deleted after use.

In our model, FS is expressed by the fact that the honesty graph is directed, i.e. the honesty of a key does not depend on keys that are derived from it. If an adversary compromises a key, keys from earlier epochs are still considered honest.

**Post-Compromise Security (PCS).** PCS was defined in the context of messaging protocols in [15] and expresses that a party can recover from compromise by introducing new key material into an existing session (and sharing that key material with group members), as long as the adversary is passive during that procedure.

PCS is expressed in the honesty graph by the fact that a key derived from two input keys (one corresponds to the old session key and one to the new key material) is honest if at least one of the input keys is honest (and the adversary doesn’t decide to compromise again).

For example, the PCS property of MLS is expressed through the fact that even if the init secret $s_{init}$ is dishonest, the joiner secret $k_{join}$ of the subsequent epoch can be honest if the commit secret $k_{commit}$ is honest.

**Freshness on Protocol-Level.** MLS is a complex protocol and our work analyses a component of it. While we leave a full analysis based on our model for future work, we now extrapolate from the key distribution freshness to the protocol-level freshness.

For example, one can conclude from the honesty graph in Figure 11 that in MLS, a single honest input key (for example a leaf key) suffices to guarantee the security of an output key. To model a protocol operation such as update on top of our model, some derived keys now need to be encrypted under public-keys (marked as depath in Figure 1). If they are encrypted under a dishonest public-key, they become known to the adversary. Thus, when deriving them, they need to be derived with an honest value so they are marked as dishonest, since our model only allows dishonest keys to be encrypted to corrupted public-keys (see the description of HPKE.ENC in Figure 7).

This procedure allows protocol-level models to encode many protocol-level attack scenarios such as active insider attacks or the double join scenario, where a party has access to secrets that are not in its direct path. Namely, the corruption mechanism of the protocol-level model would then corrupt these intermediate secrets and provide the adversary with access.

This means that a protocol-level freshness condition can’t rely on a fine-grained honesty graph such as the one in Figure 11 (which would show the corruption upon derivation of a secret, but not the reason for the corruption, e.g. a corrupt public key to which it is encrypted). Instead, it would more likely encode freshness in terms of the corruption of parties, which is what one would expect and what is the case, e.g. in the work of AJM (Figure 15 in [4]).

In summary, while our model doesn’t consider protocol-level attacks directly, we can still show that any protocol-level model built on CGKD will give a number of guarantees, for example, that the output keys of the key schedule are fresh only if either the previous epoch’s init secret is fresh or the injected PSK is fresh or the new input secret after an update operation, as well as any public key it is encrypted to.

**Adversarial Model.** The models introduced in Section V allow the adversary to corrupt all base secrets statically, and to decide to corrupt derived keys upon derivation. This corruption-upon-derivation (CUD) model simplifies the security analysis (in comparison to fully adaptive corruption) and captures the security aimed for by MLS in a meaningful way.

We chose this limitation in adversarial capabilities, as our model can’t allow the adversary to corrupt keys after they are used, as it would allow them to trivially win the distinguishing game. This problem is known as the commitment problem, initially uncovered by Nielsen [28]. Still, there is a gap between the minimal restriction required by avoiding the commitment problem and forcing the adversary to decide on corruption in the moment of derivation. It seems an interesting direction to explore whether this gap can be partially bridged by (1) postponing the onset of idealized behaviour and (2) delaying a derivation query to the point where a key is used, since else, the model does not produce any key-dependent information (and everything else can be simulated).

**Comparison to AJM** As the security model used in AJM is strictly stronger than that of ACDT, ACCK+, as well as ACJM, we now compare our model with AJM.

The first difference to AJM is that we analyze the key distribution of MLS rather than the protocol itself. Specifically, the adversary doesn’t have access to MLS protocol operations such as update, add, etc, but can instead control the inputs to the various key generation and derivation structures of the protocol directly. This means that in contrast to AJM (and traditional (key exchange) protocol security models in general), our model does not consider parties, but instead presents the adversary with access to each part of a more global protocol state (across sessions).

This allows adversaries to inject or generate key material at the various interfaces of the components of MLS and perform key derivation operations based on these keys without the strict separation of state into parties, groups or even epochs. For example, an adversary could generate an (honest) key for injection into the MLS key schedule as a PSK. That key could then be injected into the key schedule in epoch of any group and of any party and in conjunction with any other key material. The only restriction we pose is that given by the structure of the protocol: An honest key generated for injection at a given point in the key schedule cannot be used, for example, for injection at another point in the key schedule. Of course, the adversary can still inject the same (dishonest) key in multiple places.

Additionally, our work differs from AJM in that we don’t model the authentication properties of MLS and in particular don’t consider the compromise of any signature keys. Instead, our model focuses on providing the adversary with fine-
grained capabilities to compromise symmetric key material. We argue that overall, this gives the adversary more state compromise capabilities compared to AJM (with the exception of the signature key). This is because firstly, any compromise an adversary would make on a protocol level can be simulated by compromising the individual keys on a more granular level and secondly, the fine-grained access our model provides allows it to capture additional scenarios such as cross-group and cross-epoch attacks.

Finally, the findings of our analysis as detailed in this section correlate closely with those of AJM: Given our adversarial model, we establish that the keys produced by TreeKEM and the key schedule of MLS are pseudorandom if the output key is fresh. Extrapolating our freshness condition to that of a protocol-level model such as that of AJM (Figure 15 in [4]) is not straightforward, but we have shown examples earlier in this section, of how our CGKD can allow a protocol-level model to express fine-grained security guarantees comparable to those proven by AJM.

VII. THEOREMS

This section relates the security of the MLS CGKD to the security of the underlying primitives XPD, KDF, and HPKE. We first make separate security claims for the wCGKD and the security of the underlying primitives XPD, KDF, and HPKE.

Theorem 1. (wCGKD Security) For all polynomials \( d \) and all adversaries \( \mathcal{A} \) which generate trees of depth at most \( d \), it holds that

\[
\operatorname{Adv}_{\text{wCGKD}}^{\text{pke}}(\mathcal{A}) \leq \operatorname{Adv}_{\text{EXP}}^{\text{pke}}(\mathcal{A} \circ \mathcal{R}_{\text{exp}}) + \sum_{i=0}^{d-1} (\operatorname{Adv}_{\text{KDF}}^{\text{pke}}(\mathcal{A} \circ \mathcal{R}_i) + \operatorname{Adv}_{\text{HPKE}}^{\text{pke}}(\mathcal{A} \circ \mathcal{R}_{\text{hpke},i})) + \operatorname{Adv}_{\text{KDFR}}^{\text{pke}}(\mathcal{A} \circ \mathcal{R}_{\text{hpke},1}) + \operatorname{Adv}_{\text{HPKE}}^{\text{pke}}(\mathcal{A} \circ \mathcal{R}_{\text{hpke},1})
\]

where \( \text{wCGKD}^{b,d}_{\text{pke}} \) is provided in Figure 8a.

Theorem 2. (Key Schedule Security) For all polynomials \( d \) and \( t \) and all adversaries \( \mathcal{A} \) which generate trees of depth at most \( d \) and run groups for at most \( t \) epochs, it holds that

\[
\operatorname{Adv}_{\text{KDF}}^{\text{pke}}(\mathcal{A}) \leq \operatorname{Adv}_{\text{CR}}^{\text{pke}}(\mathcal{A} \circ \mathcal{R}_{\text{cr}}) + \operatorname{Adv}_{\text{KDF}}^{\text{pke}}(\mathcal{A} \circ \mathcal{R}_{\text{KDF}}) + \sum_{i=0}^{t} (\operatorname{Adv}_{\text{KDFB}}^{\text{pke}}(\mathcal{A} \circ \mathcal{R}_{\text{b},i}) + \operatorname{Adv}_{\text{HPKE}}^{\text{pke}}(\mathcal{A} \circ \mathcal{R}_{\text{hpke},i})) + \operatorname{Adv}_{\text{HPKE}}^{\text{pke}}(\mathcal{A} \circ \mathcal{R}_{\text{hpke},1})
\]

where \( \text{KDF}^{b,d,t}_{\text{pke}} \) is defined in Figure 8b.

Theorem 3. (CGKD) For all polynomials \( d \) and \( t \) and all adversaries \( \mathcal{A} \) which generate trees of depth at most \( d \) and run groups for at most \( t \) epochs, it holds that \( \operatorname{Adv}_{\text{CGKD}}^{\text{pke}}(\mathcal{A}) \)

\[
\leq \operatorname{Adv}_{\text{wCGKD}}^{\text{pke}}(\mathcal{A} \circ \mathcal{R}_{\text{wCGKD}}) + \operatorname{Adv}_{\text{KDF}}^{\text{pke}}(\mathcal{A} \circ \mathcal{R}_{\text{KDF}})
\]

where \( \text{CGKD}^{b,d,t}_{\text{pke}} \) is given in Figure 8c.

We prove Theorem 1 in Appendix VIII-A, provide the proof of Theorem 3 in Appendix B and the proof of Theorem 2 in Appendix C, D and E.
Figure 13: Values of package parameters.

(one for $b_0 = 0$ and one for $b_0 = 1$) as $G^0$ and $G^1$, if an adversary $A$ can distinguish between $G^0$ and $G^1$, then the adversary $A \circ R_{exp,i}$ can distinguish between $G_{exp}^{0,0}$ and $G_{exp}^{0,1}$. I.e., the reduction $R_{exp,i}$ is a part of code of the games $G^0$ and $G^1$ which we move into the adversary. This way of reasoning gives (a) precise definitions of reductions and (b) straightforward arguments that the simulation of the reduction is perfect. This not only makes proofs easier to verify, it also speeds up our own work since we write less reductions and changes in the code of a package immediately propagate through the entire proof and article. Publishing the code of this article should allow others to make small changes to the code without affecting the validity of the proof.

**Package Indices.** Composing different packages requires us to specify their interfaces and match them. In particular, we often compose multiple packages of the same kind in parallel, e.g., the GKDF game in Figure 25 composes $(\text{KEY}_{\text{IS}_i^{-1}})^1$, $(\text{KEY}_{\text{JS}})^0$ and $(\text{KEY}_{\text{CS}})^1$ packages in parallel which are all a variant of the $\text{KEY}^b$ package. This is analogous to classes and objects in object-oriented programming languages. A package definition is akin to a class, describing a number of method (oracles) and member variables (package state). Following that analogy, a package (or package instance), such as $(\text{KEY}_{\text{IS}_i^{-1}})^1$ is an object of that class, with the index (in subscript) distinguishing it from other objects/package instances, and with the superscript denoting member variable values (for brevity values clear from the context are omitted). Additionally, a package name suffix may be used to distinguish instances as well. If a package does not have a subscript this means there is only one instance of the package. If a package has a range (or vector) of indices (e.g. $(0..x)$) this means there are $x$ parallel instances of the package.

**Bit Indices.** The proofs of lemmas and theorems consist of game hops each of which flips a bit which we indicate by highlighting the bit (and related queries) in green in the diagrams accompanying a proof. The index of the highlighted bit corresponds to the index of the game hop. That is, we state all our theorems and lemmas in terms of flipping a single bit.

**Scoping.** Adversary and bit variables are local to a definition, theorem or diagram and unrelated to the use of a variable with the same name in other theorems and diagrams. I.e., in theorems and proofs, we split a single bit $b$ into several bits and construct multiple adversaries by composing an adversary with a reduction into $A \circ R$. Note that reduction names are globally unique in this article.

---

**Figure 14: Proof of Theorem 1**

A. Proof of Theorem 1 (wCGKD security)

The proof of Theorem 1 consists of Lemma 1 and a hybrid argument. We start with the former which shows that a TreeKEM layer with an ideal path secret can be completely idealized such that the path secret of its parent is ideal as well as the encryptions it performs.

**Lemma 1. (Layer Security)** Let $B$ be an adversary and $\text{GLAYER}_{\text{pke}}^i$ the indistinguishability game defined by Figure 9b. Then it holds that

$$\text{Adv}_{\text{GLAYER}}^i(B) \leq \text{Adv}_{\text{exp}}^i(B \circ R_{\text{exp},i}) + \text{Adv}_{\text{pke}}^i(B \circ R_{\text{pk},i})$$

where $R_{\text{exp},i}$ is defined as the grey area in Figure 12a where $R_{\text{pk},i}$ is the grey area in Figure 12b and $R_{\text{pke},i}$ as the grey area in Figure 12c.

The proof of Lemma 1 consists of three steps, each of which modifies one or two bits in the overall game. We denote

$$\text{GLAYER}_{\text{pke}}^{b_0,b_1} := \text{GLAYER}_{\text{pke}}^{b_0,b_1,b_2}$$

i.e., $\text{GLAYER}_{\text{pke}}^{b_0,b_1}$ with $b = b_0 = b_1 = b_2$. In the first proof step, we apply the expand assumption for $(b_0, b_1, b_2)$ to idealize the node secret of layer $i$ and the path secret of layer $p(i)$. We obtain the following bound:
In the third and last step of this proof we apply the HPKE derivation at the root. Therefore, as visualized in Figure 14b, we apply the expand assumption and obtain

$$\Pr\left[1 \leftarrow A \circ \text{GWCGKD}_{i}^{1,0,0,0,0} - \text{xpdk} - \text{pke}^{i}\right] \leq \text{Adv}_{\text{GWCGKD}}^{\text{xpdk}}(A \circ R_{i}),$$

where $R_{i}$ is defined as the grey area in Figure 14a. We apply this hybrid argument until $\text{KEY}_{PS0}$ is ideal. At this point the entire TreeKEM tree is ideal, except for the commit secret derived at the root. Therefore, as visualized in Figure 14b, we derive the desired bound for Theorem 1.

Recall that the $\text{KEY}_{PS}$ package represents base secrets at the lowest possible leaf layer. Thus, the package is idealized so that its values are either corrupt and chosen by the adversary or honest, randomly generated and never shared.

Due to Lemma 1, we can idealize any layer with an ideal path secret, resulting in an ideal path secret for the parent layer. Hence, we use a hybrid argument upwards through the tree, see Figure 14a. Hence, for all PPT adversaries $A$,

$$\Pr\left[1 \leftarrow A \circ \text{GWCGKD}_{i}^{1,0,0,0,0} - \text{xpdk} - \text{pke}^{i}\right] \leq \text{Adv}_{\text{GWCGKD}}^{\text{xpdk}}(A \circ R_{i}),$$

where $R_{i}$ is defined as the grey area in Figure 14a. We apply this hybrid argument until $\text{KEY}_{PS0}$ is ideal. At this point the entire TreeKEM tree is ideal, except for the commit secret derived at the root. Therefore, as visualized in Figure 14b, we apply the expand assumption and obtain

$$\Pr\left[1 \leftarrow A \circ \text{GWCGKD}_{i}^{1,0,0,0,0} - \text{xpdk} - \text{pke}^{i}\right] \leq \text{Adv}_{\text{GWCGKD}}^{\text{xpdk}}(A \circ R_{i}),$$

where $R_{i}$ is defined as the grey area in Figure 14b. Using $B = A \circ R_{i}$, we derive the desired bound for Theorem 1.

$$\text{Adv}_{\text{GWCGKD}}^{\text{xpdk}}(A) \leq \text{Adv}_{\text{GWCGKD}}^{\text{xpdk}}(A \circ R_{i}) + \sum_{i=0}^{d-1} \text{Adv}_{\text{GWCGKD}}^{\text{xpdk}}(A \circ R_{i} \circ R_{exp,i})$$

$$= \text{Adv}_{\text{GWCGKD}}^{\text{xpdk}}(A \circ R_{i} \circ R_{exp,i}) + \text{Adv}_{\text{HPKE}}^{\text{xpdk}}(A \circ R_{i} \circ R_{hpke,i}).$$

**References**


Figure 16: The MLS Key Schedule algorithm (ks) and the MLS Derive Node algorithm (dn).

APPENDIX

A. Protocol

In this appendix, we describe the algorithms update, process, and join of the MLS CGKD as well as the algorithms wudpate, wprocess, and wjoin. We first describe their syntax and then provide the algorithms.

Definition A.1 (CGKD). A continuous group key distribution (CGKD) cgkd consists of four PPT algorithms (pke.kgen, update, process, join) with inputs and outputs as specified by Figure 2 that satisfy the correctness criteria Valid Keys, Consistent Public Views and Consistent Secret Views.

Valid Keys demands that all (sk, pk) returned by the algorithms are in the range of pke.kgen. Consistent Public Views captures that different group members have consistent views on the public-keys PK_memb of each node as well as of the external public-key pk_ext. Consistent Secret Views requires that each pair of group members agrees on 1) the values of k and s_int, 2) the value of the external secret-key sk_ext and 3) the secret keys associated with the nodes the direct path of group members, i.e., the intersection of the lists SK_memb of two nodes agree.

Definition A.2 (wCGKD). A weak continuous group key distribution (wCGKD) wcgkd consists of four PPT algorithms (pke.kgen, wudpate, wprocess, wjoin) with inputs and outputs as specified by Figure 2 such that the correctness criteria Valid Keys, Consistent Public Views and Consistent Secret Views are satisfied.

The correctness criteria are the same, except that they are restricted to the inputs and outputs of wcgkd (Figure 2).

Definition A.3 (KS). A key schedule ks as takes input three symmetric keys k_int, s_int, and s_ext, a vector of labels lbl and a
update(PK_{mem}, PK_{rem}, PK_{joi}, SK_{own}, t_{own}, k_{upd}, CP, p_{ctx}, s_{ctx}, r_{ctx}, c_{ctx})

if |PK_{rem}| > 0 \lor p_{ctx} \neq \perp:
assert k_{upd} \neq \perp
(C_{cp}, PK_{upd}, PK_{mem}, SK_{mem}, k_{cs}) \leftarrow wupdate(PK_{mem}, PK_{rem}, PK_{joi}, SK_{own}, t_{own}, k_{upd}, CP, p_{ctx}, s_{ctx}, r_{ctx}, c_{ctx})

ctx \leftarrow pke.enc(pk_{ext}, s_{ctx})
\langle sk, (s_{stat}, sk_{st}), k_{ext}, ctx \rangle \leftarrow (k_{cs}, s_{stat}, sk_{st}, pke.enc(pk_{ext}, ctx))
SK_{own}[-1] \leftarrow s_{ctx}
C_{join} \leftarrow \langle \perp \rangle
for t_{local} \in PK_{joi}:
C_{join} \leftarrow pke.enc(pk_{join}, t_{local})
\langle C_{dp}, C_{join}, C_{ext}, PK_{ upd}, PK_{mem}, SK_{ own}, s_{ext}, r_{ext}, c_{ext}, CP \rangle \leftarrow \text{dopath}(PK_{mem}, t_{local}, k_{upd}, CP)
p_{ctx} \leftarrow k_{upd}
while t_{ctx} \neq \perp:
\langle ps', pk, sk \rangle \leftarrow \text{dim}(p_{ctx})
PK_{mem}[t_{local}] \leftarrow pk
SK_{own}[t_{local}] \leftarrow sk
if t_{local} \neq \perp:
for t_{local} \in CP[t_{local}]:
p_{ctx} \leftarrow PK_{mem}[t_{local}]
\langle c, \langle sk', pke.dec(pk_{ext}, ps) \rangle \rangle \leftarrow (c, \text{pke.dec}(pk_{ext}, ps))
C_{cp}[t_{local}] \leftarrow c
i_{ctx} \leftarrow \text{parent}(t_{local}, \langle PK_{mem} \rangle)
p_{ctx} \leftarrow p_{ctx}
\langle C_{dp}, PK_{mem}, SK_{ own}, ps \rangle \leftarrow \text{ret}(C_{dp}, PK_{mem}, SK_{ own}, ps)

context value ctx. It returns a secret key k_{ctx} for each lbl \in \mathbb{B}, a new key s_{ctx}, a pair (sk, pk), key k_{joi} and key k_{wext}.

For a more general definition, one might replace the sub-joist with a more general subcript.

We now provide pseudocode for MLS CGKD, MLS wCGKD and MLS KS. Recall Section III for a high-level description of update and see Figure 17 for its code. The blue and pink inputs are optional. The code colored in the corresponding color is only executed if the corresponding input is provided. I.e., encryption to the external public-key pk_{ext} is only performed if there is an external public-key pk_{ext} in the input, modeling and external add operation. Note that there is an if condition which requires that if pk_{ext} is given, then also k_{upd} must be given. This is because an external add operation requires adding new key material as the external updater does not know the current key material. In turn, for an internal add operation, if it is performed as add-only operation, then k_{upd} would be empty. In this case, the wupdate procedure (Figure 17) does not run the encryption to the co-path, encoded as the wopath procedure.

The algorithms process and join update their key material consistently with update. I.e., the ciphertexts C_{cp} and the list of changes to the public-key information PK_{rem}, PK_{joi} and PK_{upd} suffice for process to derive the same values as update. Similarly, the ciphertexts C_{join} and the view of the entire new tree PK_{mem} suffice to allow the join procedure to derive the same values. See Figure 17 for their code.

B. Proof of Theorem 3 (CGKD security)

We prove Theorem 3 using Theorem 1 and Theorem 2. Recall (see Figure 8a) that, if b = b_0 = b_1, then
\text{GCGKD}_{b_0, b_1, d, t}^{xp, kdf, pke} = \text{GCGKD}_{b_0, b_1, d, t}^{xp, kdf, pke}.

In the first step of this proof we modify b_0 from 0 to 1 and reduce to GWCCKD. For all PPT adversaries A, we obtain
\text{Adv}_{\text{GCGKD}}^{xp, kdf, pke}(A, R_{\text{CGKD}}),
where R_{\text{CGKD}} is defined as the gray area in Figure 15(a).
Next, we modify b_1 from 0 to 1 and reduce to GKS. We obtain
\text{Adv}_{\text{GCGKD}}^{xp, kdf, pke}(A, R_{\text{KS}}),
where R_{\text{KS}} is defined as the gray area in Figure 15(b).
Hence, we obtain the desired bound for Theorem 3.

C. Proof of Theorem 2 (KS security)

This appendix proves Theorem 2. We first idealize collision-resistance for all primitives (unique inputs translate into unique outputs), then idealize the two KDFs of the key schedule based on the commit secret and the external secret globally for all epochs (since the commit secret and the external secret do not have an epoch) and then finally apply a hybrid argument over the epochs which is captured by Lemma 3. 
KEY_IS_0 represents the initial init secret and is ideal from the start, i.e., it samples honest values independently at random and allows the adversary to set corrupt values. Similarly, the pre-shared keys (KEY_PSK) are ideal from the start as they are exchanged out-of-bound. Lastly, note that the public-private key-pairs (KEY_NKP) and commit secrets (KEY_CS) are ideal form the start, which follows from Theorem 1.

Figure 10a depicts GGKS_{b_0, b_1, d, t} for trees up to depth d and groups that run for up to t epochs. For $b = b_0 = b_1$, we define GGKS_{b_0, b_1, d, t}. Here, $b_0$ corresponds to global proof steps (collision-resistance and KDFs), and $b_1$ corresponds to epoch-wise proof steps. Note that the second $G$ in $GGKS$ stands for global.

We now state Lemma 2 and Lemma 3, then show that they imply Theorem 2 and then prove Lemma 2 and Lemma 3.

**Lemma 2.** For all polynomials $d$ and $t$ and all adversaries $A$ which generate trees of depth at most $d$ and run groups for at most $t$ epochs, it holds that $Adv_{GGKS}^{0..d, t}_{\text{epd,KDF,kpe}}$: $$(A) := \left| \Pr \left[ 1 \leftarrow A \circ \text{GGKS}^{0..d, t}_{\text{epd,KDF,kpe}} \right] - \Pr \left[ 1 \leftarrow A \circ \text{GGKS}^{0..d, t}_{\text{epd,KDF,kpe}} \right] \right| \leq Adv_{\text{CR}}^{0..d, t}_{\text{epd,KDF}(A \circ \text{KDFL})} + \text{Adv}_{\text{KDFL}}^{0..d, t}_{\text{epd,KDF}(A \circ \text{KDFL})}$$

Recall that GGKS_{b_0, b_1, d, t} is defined in Figure 10a. Analogously, we also use the notation $Adv_{\text{epd,KDF,kpe}}^{0..d, t}_{\text{epd,KDF,kpe}}$: $$(A) := \left| \Pr \left[ 1 \leftarrow A \circ \text{GGKS}^{0..d, t}_{\text{epd,KDF,kpe}} \right] - \Pr \left[ 1 \leftarrow A \circ \text{GGKS}^{0..d, t}_{\text{epd,KDF,kpe}} \right] \right|$$

**Lemma 3.** (Epoch Security) Let $B$ be an adversary and $GEPOCH_{\text{d, t}, i}$ the indistinguishability game defined by Figure 25. Then $\forall d, t, i \in \mathbb{N}$: $0 \leq i \leq t$, $Adv_{\text{epd,KDF,kpe}}^{0..d, t}_{\text{epd,KDF,kpe}}(B) :=$ $Pr \left[ 1 \leftarrow B \circ \text{GEPOCH}^{0..d, t}_{\text{epd,KDF,kpe}} \right] - Pr \left[ 1 \leftarrow B \circ \text{GEPOCH}^{0..d, t}_{\text{epd,KDF,kpe}} \right]$ $\leq Adv_{\text{KDF}}^{0..d, t}_{\text{EP-}b_0, d_0, t_0}(B) + Adv_{\text{KDF}}^{0..d, t}_{\text{EP-}b_1, d_1, t_1}(B) + Adv_{\text{KDF}}^{0..d, t}_{\text{EP-}b_2, d_2, t_2}(B) + Adv_{\text{KDF}}^{0..d, t}_{\text{EP-}b_3, d_3, t_3}(B) + Adv_{\text{KDF}}^{0..d, t}_{\text{EP-}b_4, d_4, t_4}(B) + Adv_{\text{KDF}}^{0..d, t}_{\text{EP-}b_5, d_5, t_5}(B) + Adv_{\text{KDF}}^{0..d, t}_{\text{EP-}b_6, d_6, t_6}(B)$

where $R_{b_i, d_i}^{0..i}$ is defined as the grey area in Figure 24, $R_{b_2, d_2}^{0..i}$ as the grey area in Figure 25, $R_{b_4, d_4}^{0..i}$ as the grey area in Figure 26, and $R_{b_5, d_5}^{0..i}$ as the grey area in Figure 28.

**Proof of Theorem 2.** We first apply Lemma 2 and then apply a hybrid argument over the $t$ epochs using that for $0 \leq i \leq t-1,
D. Proof of Lemma 2

In order to prove Lemma 2, we introduce game \( \text{GPGKS}^{b_0,b_1,b_2,d,t}_{\text{spdkdf,kpe}} \) depicted in Figure 20, where the \( P \) in GPGKS stands for proof. The following equalities hold:

\[
\text{GPGKS}^{0,0,d,t}_{\text{spdkdf,kpe}} = \text{GPGKS}^{0,0,d,t}_{\text{spdkdf,kpe}}
\]
\[
\text{GPGKS}^{1,1,d,t}_{\text{spdkdf,kpe}} = \text{GPGKS}^{1,0,d,t}_{\text{spdkdf,kpe}}
\]

For collision-resistance, we can construct a reduction \( \mathcal{R}_{\text{psk}} \) such that the distinguishing advantage between \( \text{GPGKS}^{0,0,d,t}_{\text{spdkdf,kpe}} \) and \( \text{GPGKS}^{1,0,d,t}_{\text{spdkdf,kpe}} \) can be turned into a collision against one of the underlying primitives. This is possible because (a) the initial commit secret, PSK, and level 0 init secrets are unique and (b) we have domain separation between the different epochs due to the inclusion of the group context in the KDFs. Thus, we can argue inductively that

\[
\mathbb{Pr}[1 \leftarrow A \circ \text{GPGKS}^{0,0,d,t}_{\text{spdkdf,kpe}}] - \mathbb{Pr}[1 \leftarrow A \circ \text{GPGKS}^{1,0,d,t}_{\text{spdkdf,kpe}}] \leq \text{Adv}_{\text{spdkdf,kpe}}(A \circ \mathcal{R}_{\text{psk}})
\]

After idealizing collision-resistance, we can now reduce to the LKDF security of the commit secret, since the respective second inputs are unique. We thus obtain that

\[
\mathbb{Pr}[1 \leftarrow A \circ \text{GPGKS}^{1,0,d,t}_{\text{spdkdf,kpe}}] - \mathbb{Pr}[1 \leftarrow A \circ \text{GPGKS}^{1,1,0,d,t}_{\text{spdkdf,kpe}}] \leq \text{Adv}_{\text{KDFL}}(A \circ \mathcal{R}_{\text{cs}})
\]

where \( \mathcal{R}_{\text{cs}} \) is defined in Figure 21. Analogously,

\[
\mathbb{Pr}[1 \leftarrow A \circ \text{GPGKS}^{1,1,0,d,t}_{\text{spdkdf,kpe}}] - \mathbb{Pr}[1 \leftarrow A \circ \text{GPGKS}^{1,1,1,0,d,t}_{\text{spdkdf,kpe}}] \leq \text{Adv}_{\text{KDFL}}(A \circ \mathcal{R}_{\text{psk}})
\]

where \( \mathcal{R}_{\text{psk}} \) is defined in Figure 22. We obtain

\[
\mathbb{Pr}[1 \leftarrow A \circ \text{GPGKS}^{0,0,d,t}_{\text{spdkdf,kpe}}] - \mathbb{Pr}[1 \leftarrow A \circ \text{GPGKS}^{1,0,d,t}_{\text{spdkdf,kpe}}] \leq \text{Adv}_{\text{spdkdf,kpe}}(A \circ \mathcal{R}_{\text{psk}}) + \text{Adv}_{\text{KDFL}}(A \circ \mathcal{R}_{\text{psk}}),
\]

which concludes the proof of Lemma 2.

E. Key Schedule Epoch

We now prove Lemma 3 which captures the security of a single epoch of the key schedule, a core argument in the proof of Theorem 2. We show that when an MLS epoch relies on an ideal internal secret \( s_{int} \) derived in a previous epoch or an ideal PSK \( s_{ext} \) or an ideal commit secret \( k_{cs} \), then the current epoch provides security guarantees, too, for the new \( s_{int} \) derived in the current epoch, all group secrets for the current epoch and for the current external public-key encryption keys.
Figure 24: Step 1 of idealizing epoch $i$. We mark $R_{2,i}^{ks}$ in grey.

The proof of Lemma 3 consists of six steps, each of which flips one or more bits in the overall game. See Figure 25 for the definition of $\text{GEOCH}_{0..d}^{k,\text{d,kdf, pke}}$ and note that $\text{GEOCH}_{0..d}^{k,\text{d,kdf, pke}} = \text{GEOCH}_{0..d}^{k,\text{d,kdf, pke}}$ if $b_0 = b_1 = b_2 = b_3 = b_4 = b_5$.

In the first step of this proof we use the KDFR assumption to idealize the $\text{KEY}_JS$ package of epoch $i$. We obtain

$$\Pr \left[ 1 \leftarrow A \circ \text{GEOCH}^{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0} \text{d,kdf, pke} \right] - \Pr \left[ 1 \leftarrow A \circ \text{GEOCH}^{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0} \text{d,kdf, pke} \right] \leq \text{Adv}^\text{kdf}_{\text{KDFR}}(A \circ R_{2,i}^{ks}).$$

Now, in the second game hop, we use the HPKE assumption for $n = 1$ and $m = d$ to idealize the first HPKE package of epoch $i$ and to remove access to the joiner secret. We obtain

$$\Pr \left[ 1 \leftarrow A \circ \text{GEOCH}^{1,0,0,0,0,0,0,0,0,0,0,0,0,0,0} \text{d,kdf, pke} \right] - \Pr \left[ 1 \leftarrow A \circ \text{GEOCH}^{1,1,0,0,0,0,0,0,0,0,0,0,0,0,0} \text{d,kdf, pke} \right] \leq \text{Adv}^\text{pke}_{\text{HPKE}}(A \circ R_{2,i}^{ks}).$$

In the third step of this proof we use the KDFR assumption to idealize the $\text{KEY}_ES$ package of epoch $i$. This is possible since there is no GET query on the joiner secret anymore. We obtain the bound

$$\Pr \left[ 1 \leftarrow A \circ \text{GEOCH}^{1,1,0,0,0,0,0,0,0,0,0,0,0,0,0} \text{d,kdf, pke} \right] - \Pr \left[ 1 \leftarrow A \circ \text{GEOCH}^{1,1,1,0,0,0,0,0,0,0,0,0,0,0,0} \text{d,kdf, pke} \right] \leq \text{Adv}^\text{kdf}_{\text{KDFR}}(A \circ R_{3,i}^{ks}).$$

Figure 25: Step 2 of idealizing epoch $i$. We mark $R_{2,i}^{ks}$ in grey.

Figure 26: Step 3 of idealizing epoch $i$. We mark $R_{3,i}^{ks}$ in grey.

Figure 27: Step 4 of idealizing epoch $i$. We mark $R_{4,i}^{ks}$ in grey.
In the sixth and final step of the epoch proof we use the HPKE assumption for \( n = 1 \) and \( m = 1 \) to idealize the second HPKE package of epoch \( i \). We obtain

\[
\Pr \left[ 1 \leftarrow A \circ \text{GEO}^{1,1,1,1,0,d,t,i} \right] - \Pr \left[ 1 \leftarrow A \circ \text{GEO}^{1,1,1,1,0,d,t,i} \right] \leq \text{Adv}^{\text{HPKE}}(A \circ R_{6,i}).
\]

Hence,

\[
\text{Adv}^{\text{EPOCH}}(A) \leq \text{Adv}^{\text{KDF}}(A \circ R_{6,i}) + \text{Adv}^{\text{PKE}}(A \circ R_{6,i})
\]

\[
+ \text{Adv}^{\text{KDF}}(A \circ R_{6,i}) + \text{Adv}^{\text{EXP}}(A \circ R_{6,i})
\]

\[
+ \text{Adv}^{\text{DKP}}(A \circ R_{6,i}) + \text{Adv}^{\text{HPKE}}(A \circ R_{6,i}),
\]

and Lemma 3 holds.

In the fourth step of this proof we use the XPD assumption with \( n = 2 + |GR| \) to idealize all current group secrets in \( \text{KEY}_\text{GR} \), the init secret in \( \text{KEY}_\text{IN} \) and the seed in \( \text{KEY}_\text{EX} \) (later used for computing the external public-key). We obtain

\[
\Pr \left[ 1 \leftarrow A \circ \text{GEO}^{1,1,1,1,0,d,t,i} \right] - \Pr \left[ 1 \leftarrow A \circ \text{GEO}^{1,1,1,1,0,d,t,i} \right] \leq \text{Adv}^{\text{XPD}}(A \circ R_{5,i}).
\]

In the fifth step of this proof, we lose the DKP advantage to idealize the \( \text{PKE}_\text{EX} \) package of epoch \( i \), and we obtain

\[
\Pr \left[ 1 \leftarrow A \circ \text{GEO}^{1,1,1,1,0,d,t,i} \right] - \Pr \left[ 1 \leftarrow A \circ \text{GEO}^{1,1,1,1,0,d,t,i} \right] \leq \text{Adv}^{\text{DKP}}(A \circ R_{5,i}).
\]