UNIVERSITY OF TARTU
Institute of Computer Science
Computer Science Curriculum

Eric Cornelissen

Cryptographic Analysis of the Message Layer Security Protocol in the Static Corruption Model

Master's Thesis (30 ECTS)

Supervisor: Chris Brzuska, PhD
Supervisor: Dominique Unruh, PhD
Instructor: Konrad Kohbrok, MSc

Espoo, May 15, 2020

Cryptographic Analysis of the Message Layer Security Protocol in the Static Corruption Model

Abstract:
Existing cryptographic protocols achieve a range of security guarantees such as secrecy and authentication. However, most protocols are designed for one-to-one communication and protocols for group communication are less common, often less efficient, and typically provide fewer security guarantees. This is because group communication poses unique challenges, such as coordinated key updates and changes to group membership, that complicate the protocol design. Still, group communication is common in messaging applications and often security is sacrificed for efficiency.

The IETF created a working group with the goal of bridging this gap by developing a standard for a continuous asynchronous key-exchange protocol for dynamic groups that is secure and remains efficient for large group sizes. This thesis provides a cryptographic analysis of TreeKEM and the key schedule present in draft 8 of the Message Layer Security (MLS) protocol RFC. The analysis is carried out using the State Separating Proofs methodology [BDLF+18].

We show that both the keys produced by TreeKEM and the key schedule of MLS are pseudorandom in the static adversarial model given standard assumptions on Pseudorandom Functions, Key Derivation Functions, and Public-Key Encryption, giving concrete security bounds for both.

Keywords:
Protocol Analysis, State Separating Proofs, Message Layer Security, TreeKEM

CERCS: P170 Computer science, numerical analysis

Sõnumikihi turvaprotokolli krüptograafiline analüüs staatilise korruptsiooni mudelis

Lühikokkuvõte:
Olemasolevad krüptoprotokollid pakuvad mitmesuguseid turvagarantiisid, näiteks salastatus ja autentimine. Enamik protokolle on mõeldud üks-ühele suhtluseks ja rühmasuhtluse protokollid on vähem levinud, sageli vähem tõhusad ja pakuvad tavaliselt vähem turvagarantiid. Selle põhjuseks on see, et rühmasuhtlus tekitab ainulaadseid väljakutseid, näiteks koordineeritud võtmevärskendused ja grupiliikme muudatused, mis raskendavad protokolli kujundamist. Sellegipoolest on rühmasuhtlus sõnumsiderakendustes tavaline ja tõhususe nimel ohverdatakse sageli turvalisust.

IETF lõi töörühma eesmärgiga see lõhe ületada, töötades välja dünaamiliste rühmade jaoks pideva asünkroonse võtmevahetuse protokolli standardi, mis on turvaline ja efek-

tiivne ka suurte rühmade korral. See lõputöö pakub välja TreeKEM ja MLS-protokolli RFC 8. eelnõus sisalduva võtmegraafiku krüptograafilise analüüsi. Analüüs viiakse läbi riiki eraldavate tõendite metoodika [BDLF+18] abil.

Näitame, et nii TreeKEM toodetud võtmed kui ka MLS-i võtmegraafik on staatilise võistlusmudeli korral pseudo-juhuslikud, arvestades pseudo-haruldaste funktsioonide, võtme tuletusfunktsioonide ja avaliku võtme krüptimise standardseid eeldusi, mis annavad mõlemale konkreetsed turbepiirid.

Võtmesõnad:
Protocol Analysis, State Separating Proofs, Message Layer Security, TreeKEM

CERCS: P170 Arvutiteadus, arvutusmeetodid

Contents

Chapter 1

Introduction

It is generally expected that modern-day messaging applications provide end-to-end security, guaranteeing secrecy of message contents to anyone but the intended recipient. This expectation has been made a reality over the course of 25 years through the design of cryptographic protocols which nowadays they guarantee secure communication, potentially even after a participant is compromised. Moreover, such protocols are commonly used in practice.

This is due in part to two-party key-exchange protocols, for which various security notions have emerged. *Key Indistinguishability* [dSGFW19, KMVOV96], which informally ensures that keys are known only to the communicating participants, guarantees a secret key and *Entity Authentication* [BR93, KMVOV96] guarantees that only the intended recipient can read messages. More recently, a variant of key indistinguishability called *Forward Secrecy* (FS) was defined, which requires old sessions to be protected if a new session is breached [KMVOV96].

The combination of key indistinguishability and entity authentication results in shared keys with certain security properties. In a two-phase protocol design, this key is subsequently used to encrypt the communication between parties. The communication may then be considered secure w.r.t. the key's properties.

The *asynchronous* nature of modern messaging applications, where a recipient might be offline when a message is sent, brings additional challenges to the table for protocol designers. This is in contrast to standard key-exchange protocols where participants are assumed to be online. Moreover, messaging sessions are continuous as they can last for several months or even years, whereas e.g. a phone call typically lasts at most an hour. As such, non-interactive protocols were invented that perform key-exchange and entity authentication continuously for long periods.

With the introduction of *continuous* key-exchange, another variant of key indistinguishability was introduced named *Post-Compromise Security* (PCS) [CGCG16, KPT01], which informally ensures security for subsequent communication. Both FS and PCS have been achieved in newer protocols such as the *Signal protocol* (formerly *TextSecure*) due to

the use of the double ratchet protocol (formerly *Axolotl*) [CGCD$^+$17, CGCG16, PM16].

The continuous and asynchronous two-party case can be extended to the group setting where the objective is to obtain a shared secret known only to a predetermined group of parties, called *members*, that are authenticated to each other [ACDT19]. Early on, group key-exchange protocols that are straightforward extensions of two-party protocols were common. For example, the protocol by Burmester-Desmedt [BD94] uses point-to-point channels to share a secret key amongst all group members. However, such protocols are 1) designed for static groups; 2) not continuous; 3) synchronous, as all group members need to be online to derive the keying material; and 4) inefficient as groups grow larger, with a complexity quadratic in the size of the group due to the point-to-point design.

In [SSDW88] a group key-exchange protocol is introduced based on an unbalanced, Diffie-Hellman shares-based, left-leaning binary tree to obtain a shared secret for a group (illustrated in Figure 1.1). In this tree, the leaves contain per-member private Diffie-Hellman shares which are recursively combined to obtain a group secret at the root of the tree. In [KPT01] this protocol was altered to allow for dynamic groups by adding new members as new leaves in the tree. It also introduced the notions of *Weak Forward Secrecy* and *Weak Post-Compromise Security* which require, respectively, that new members cannot discover past group secrets and removed members cannot discover future group secrets. Later, in [WGL00] the efficiency of adding and removing members from a group was improved by enforcing the use of a balanced binary tree.

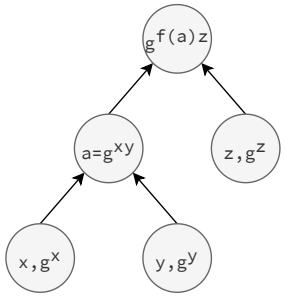


Figure 1.1: (Where f encodes a as a valid exponent.)

However, none of the protocols mentioned provides a way to update ephemeral keys. [CGCG$^+$18] addresses this limitation by introducing an *Asynchronous Ratcheting Tree* (ART) which enables group members to update their leaf key and thereby change the group secret asynchronously. This improves both FS and PCS within groups in comparison to earlier protocol.

The *Message Layer Security* (MLS) Working Group of the Internet Engineering Task Force (IETF) aims to create a new and efficient protocol for continuous asynchronous group key-exchange and messaging [BBM$^+$19]. To achieve this, a novel group key-exchange protocol that is based on TreeKEM [BBR18], which is inspired by ART, is part of the current draft.

In short, TreeKEM is 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 with subsets of the group. The secret value at the root node of the tree is a secret shared by the entire group, which is used in the key schedule of

MLS. To update one's keying material, add a new member, or remove a member the key pairs and values of a leaf-to-root path are updated by one member and shared efficiently. TreeKEM is explained in more details in Chapter 3.

MLS Analyses The MLS RFC has been the subject of several research efforts [ACC+, ACDT19, CHK19, Wei19]. In [ACC+] an alternative to TreeKEM is proposed, named *Tainted* TreeKEM, which replaces blank nodes, explained in Section 3.1, with so-called *tainted* nodes. In contrast to a blank node, the keys of a tainted node are still used. A tainted node will be blank after the entity that tainted the node updates their keying material. In doing so, depending on the distribution of add and remove operations, the efficiency of updates is increases compared to TreeKEM.

Relatedly, [Wei19] proposes an alternative version of TreeKEM, named *Causal* TreeKEM, that does not require a strict total order of messages. Where the MLS RFC assumes a trusted server that can enforce ordering of messages because key updates overwrite existing keys, Causal TreeKEM does not have this requirement because it uses keying material that can be combined in an arbitrary order. However, this requires keying material that is associative and commutative, such as Diffie-Hellman shares. MLS instead opts for higher modularity with a KEM approach, where the underlying encryption scheme can be changed if needed. Specifically, TreeKEM can become post-quantum secure by changing its KEM strategy, which is not possible in causal TreeKEM.

In [CHK19], a security analysis of MLS reveals that Post-Compromise Security is not guaranteed if a client is a member of multiple groups (although PCS is guaranteed for a single group). This is a result of the updating mechanism, which affects the ephemeral keying material of one group only, not that of other groups nor, crucially, the long-term keying material. The implication is that an adversary can impersonate the compromised client in all non-recovered groups, including those not yet started.

Relatedly, [ACDT19] shows that the Forward Secrecy guarantees of MLS are relatively weak. An update to the keying material of one client will leave the group vulnerable w.r.t. FS from other corrupted clients. Therefore, in the worst case, FS is only achieved if, every group member updates their keying material. To mitigate this shortcoming it is proposed to use a PKE scheme in TreeKEM where keys are updated with every encryption and decryption operation.

Contributions To provide confidence about the security level claimed in the RFC, the protocol needs to be subjected to cryptographic analysis (as illustrated by preceding paragraphs). This thesis aims to contribute to the drafting process by performing a cryptographic analysis on draft 8 of the MLS specification using the State Separating Proofs methodology [BDLF+18]. The scope of the analysis is limited to the static-corruption model, focussing on TreeKEM and the key schedule of MLS, and giving a concrete security bound for both.

The results show that, under natural assumptions on Key Derivation Functions and Public-Key Encryption schemes, TreeKEM achieves its primary security goal (Chapter 6); to obtain a common shared secret for a group which cannot be guessed by an outsider. Additionally, the results show that, under natural assumptions on Pseudorandom Functions and Key Derivation Functions, the key schedule generates pseudorandom key material upon updates (Chapter 7).

Additionally, the present work provides a novel definition of a continuous asynchronous group key agreement protocol notion (Chapter 4), along with an MLS implementation of that notion, as well as a novel Public Key Encryption assumption in the State Separating Proofs methodology based on IND-CCA security [An01] (Section 5.3).

Overview Chapter 2 introduces the notation, terminology, definitions, and methodology used in this thesis. Chapter 3 explains the novel components of the MLS RFC, covering TreeKEM (Section 3.1) and the key schedule (Section 3.3). Chapter 4 continues by describing the adversarial model as well as the group key agreement protocol syntax used as a basis for the present work. Next, Chapter 5 present the cryptographic assumptions upon which the security proofs of this thesis are based. Then, Chapter 6 and Chapter 7 give a security proof of TreeKEM and the key schedule components of MLS, respectively, in the specified security model. Finally, Chapter 8 provides a discussion and other concluding remarks.

Remark The present work considers draft 8 of the MLS RFC [BBM+19]. This is the latest complete version of the RFC as of May 14, 2020. Where draft 8 is incomplete or ambiguous, modifications leading up to draft 9 of the RFC have been considered.

Chapter 2

Preliminaries

This chapter describes the notation, terminology, definitions, and methodology used in this thesis.

2.1 Notation

Let $(x \cdot y)$ denote the multiplication of x and y, $(x \oplus y)$ the bit-wise exclusive or of x and y, and $(x \| y)$ the concatenation of x and y. Let $(a, b, \ldots)$ be a tuple of the values a, b etc., $\{a, b, \ldots\}$ a set of the values a, b etc., and $[a, b, \ldots]$ a list of values a, b etc. Let $\{0,1\}^\lambda$ be the set of all bitstring of length λ and $\{0,1\}^*$ the set of all finite bitstrings. Let $\emptyset$ denote the empty set (i.e. $\emptyset := \{\}$) and $[\,]$ an empty list. If T is a table, let $\mathrm{T}[x]$ be the value stored in T for handle x. By default the value for any handle in a table is $\perp$. For a tuple, set, list, (bit)string, or table x, let $|x|$ be its size or length.

Let $x \leftarrow y$ denote that x is assigned the value of y. If f is a function, then $x \leftarrow f(y)$ denotes that x is assigned the value of evaluating f on y. If $\mathcal{X}$ is a set, then $x \leftarrow\!\!\$\ \mathcal{X}$ denotes sampling a uniformly random value from $\mathcal{X}$ and assigning it to x. If X is a randomised or probabilistic algorithm, then $x \leftarrow\!\!\$\ \mathrm{X}()$ denotes running X with uniform coin tosses and assigning the result to x. If l is a list and x is a value, then $l \leftarrow\!\!/\!/\ x$ denotes appending x to the end of l.

Let $x := y$ denote that x is defined as y, and let $x \equiv y$ denote that x is equivalent to y. In particular, if $\mathcal{A}$ and $\mathcal{B}$ are algorithms, then $\mathcal{A} \overset{\text{code}}{\equiv} \mathcal{B}$ denotes that $\mathcal{A}$ and $\mathcal{B}$ are code equivalent, i.e. the code is equal up to renaming of variables.

Pseudocode For pseudocode the notion of a set is generalised to that of an object. In short, an object is an index-value store of arbitrary size. For convenience, $x \leftarrow \{y\}$ denotes storing the value of y in an object under an index named y. Then, $z \leftarrow x.y$ denotes assigning the value stored under index y in x to z.

The present work assumes a functional programming paradigm for pseudocode, leading to functions without side effects. Most notably, this means that changes to an object create a "shadow" object. I.e. the original object is not changed in memory and instead a new object with the change is placed in memory.

In pseudocode, $[x = y]$ (resp. $[x \neq y]$) is used to check for equality (resp. inequality) between x and y returning `true` (1) or `false` (0). $(a, b, \ldots) \leftarrow t$ denotes decomposing the tuple t into $a, b, \ldots$ based on index and $\{a, b, ...\} \leftarrow s$ denotes decomposing an object s based on index name.

The symbol $\perp$ is used to represent a missing value (similar to `null`). The symbol $\diamond$ is used to represent a failed computation (similar to an error), e.g. due to illegal parameters.

Logic notation for *and* ($\wedge$), *or* ($\vee$), and *negation* ($\neg$) is used in pseudocode, rather than the conventional notation used in popular programming languages: `&&`, `||`, and `!` respectively.

2.2 Terminology

This section introduces terminology specific to MLS (Section 2.2.1) and terminology regarding trees in the Computer Science context (Section 2.2.2).

2.2.1 MLS

Following the MLS RFC [BBM+19] a *Client* is an agent (person or device) that uses a group key-exchange protocol to establish shared cryptographic state with other Clients. A *Group* is a collection of one or more Clients with shared cryptographic state. A *Member* of a Group is a Client that is part of that Group. An *epoch* is a number indicating how many times the keying material of a Group has been updated, i.e. an update to the keys of a Group advances the epoch of that Group by one.

A *Proposal* is a message containing a suggestion to update either the Group structure (add or remove a Member) or the ephemeral keys of the Member creating the Proposal. A *Commit* is a message containing a collection of Proposals for one Group that the Members of that Group may[1] use to advance the group state and secrets to the next epoch. A *Welcome* message is a message sent to Clients added to a Group so they can derive the Group secrets and start participating in that Group starting from the epoch after the Proposal to add them was made.

More details on the functionality related to this terminology can be found in Chapter 3.

[1] Multiple Commits may be suggested for one epoch by different members, but in the end, only a single commit can be used to advance the group state.

2.2.2 Trees

A *tree* is a special instance of a graph that is 1) fully connected and 2) contains no cycles. The *root* (node) of a tree is the only node without an incoming edge. Every other node has exactly one incoming edge, the node at the other end of this edge is called the *parent* (node). The *ancestors* of a node are its parent, its parent's parent, etc. until the root. The nodes at outgoing edges of a parent node are called *children* (or *child* nodes). A node without any children is called a *leaf* (node). A *subtree* of any tree is the tree acquired when considering any node in the tree as the root of a new tree. The *height* of a tree is equal to the maximum number of ancestors of any node in the tree. The height of a tree consisting of only a root node has height zero.

A *binary tree* is a special instance of a tree where every parent node has either zero or two child nodes. A *balanced binary tree* is a special instance of a binary tree where the difference in height between the left and right subtree of the root never exceeds one. A *left-balanced binary tree* is a special instance of a balanced binary tree where the left subtree of a node is always filled before the right subtree of a node.

2.3 Cryptographic Primitives

The MLS RFC makes use of a variety of cryptographic primitives. This section introduces the syntax and describes the functionality of each of the cryptographic primitives used in the present work.

Hash A Hash Function, H, is a function $y \leftarrow \mathrm{H}(x)$ that converts an arbitrary size bitstring into a fixed-length bitstring.

MAC A Message Authentication Code (MAC) scheme μ is a tuple of two functions $\mu := (\mathrm{tag}, \mathrm{ver})$, one function to generate message tags and one function to verify message tags.

- $t \leftarrow\!\!\$\ \mu.\mathrm{tag}(\mathsf{k}, m)$: The tagging algorithm is a randomised algorithm that generates tags t for a plaintext m for a given key k.
- $b \leftarrow \mu.\mathrm{ver}(\mathsf{k}, t, m)$: The verification algorithm is a deterministic algorithm that verifies if a tag t corresponds to a plaintext m for a given key k. The output value is a bit $b \in \{0, 1\}$.

Moreover, a MAC scheme needs to satisfy correctness. I.e.:

$$\forall \mathsf{k} \in \{0,1\}^{\lambda}\ \forall m \in \{0,1\}^{*} : [\mu.\mathrm{ver}(\mathsf{k}, \mu.\mathrm{tag}(\mathsf{k}, m), m) = 1],$$

where λ is the size of keys used by the MAC scheme.

Pseudorandom Function A Pseudorandom Function (PRF) is a deterministic keyed function $y \leftarrow \varphi(\mathsf{k}, x)$ whose behaviour is indistinguishable from a random function. In this thesis, the notion of a *Dual* Pseudorandom Function (DPRF) is a adopted. In this adopted notation, the role of input and key are interchangeable.

In particular, if either key or input are uniformly random and unknown to the adversary, the output of the function is pseudorandom. This notion is formalised in Section 5.1.

Key Derivation Function A Key Derivation Function (KDF) κ is a function that "expands" a bitstring into fixed-length bitstrings for a given label [Kra10]. A KDF, $y \leftarrow \kappa(x, L)$, is a deterministic algorithm that takes as input a bitstring and a "label" and outputs a single bitstring. Labels allow multiple keys to be derived from the same input value x.

Intuitively, the output value y should look random if x is secret (the label is typically a publicly known value) and has high entropy. This notion is formalised in Section 5.2.

Remark In the RFC, following [Kra10], a KDF is considered as a primitive that combines the function of a PRF (extraction) and KDF (expansion).

Symmetric Encryption An Authenticated Encryption with Associated Data (AEAD) scheme ξ is a tuple of two functions $\xi := (\text{enc}, \text{dec})$, one function to encrypt plaintexts and one function to decrypt ciphertexts.

- $c \leftarrow\!\!\$\ \xi.\text{enc}(\mathsf{k}, n, m, \alpha)$: The encryption algorithm is a randomised algorithm that takes a secret key k, nonce n, plaintext m, and Additional Authenticated Data (AAD) α, and outputs a ciphertext c.
- $m \leftarrow \xi.\text{dec}(\mathsf{k}, n, c, \alpha)$: The decryption algorithm is a deterministic algorithm that takes a secret key k, nonce n, ciphertext c, and AAD α, and outputs the original plaintext m or the $\perp$ symbol if the decryption fails.

Moreover, an AEAD scheme needs to satisfy correctness. I.e.:

$$\forall \mathsf{k} \in \{0,1\}^{\lambda_1}\ \forall n \in \{0,1\}^{\lambda_2}\ \forall m \in \{0,1\}^*\ \forall \alpha \in \{0,1\}^* \\ : \Pr[\xi.\text{dec}(\mathsf{k}, n, \xi.\text{enc}(\mathsf{k}, n, m, \alpha), \alpha) = m] = 1,$$

where λ_1 is the size of keys and λ_2 is the size of nonces used by the AEAD scheme.

Public Key Encryption Following [An01], a Public Key Encryption (PKE) scheme ζ is a tuple of three functions $\zeta := (\text{gen}, \text{enc}, \text{dec})$, one function to generate key pairs, one function to encrypt plaintexts using a public key, and one function to decrypt ciphertexts using a secret key.

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \zeta.\mathsf{gen}()$: The key generation algorithm is a randomised algorithm that outputs a key pair $(\mathsf{sk}, \mathsf{pk})$. Optionally, the algorithm can be made deterministic by providing a seeding value s as input ($\zeta.\mathsf{gen}(s)$).
- $c \leftarrow\!\!\$\ \zeta.\mathsf{enc}(\mathsf{pk}, m)$: The encryption algorithm is a randomised algorithm that generates a ciphertext c from a plaintext m for a given public key pk.
- $m \leftarrow \zeta.\mathsf{dec}(\mathsf{sk}, c)$: The decryption algorithm is a deterministic algorithm that takes as input a private key sk and a ciphertext c, and outputs a plaintext m or the $\perp$ symbol if the decryption fails.

Moreover, a PKE scheme needs to satisfy correctness. I.e.:

$$\forall(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \zeta.\mathsf{gen}()\ \forall m \in \{0,1\}^* : \Pr[\zeta.\mathsf{dec}(\mathsf{sk}, \zeta.\mathsf{enc}(\mathsf{pk}, m)) = m] = 1.$$

Intuitively, a PKE scheme should produce ciphertexts that do not leak any information about the plaintext to anyone without knowledge of the secret key corresponding to the public key used for encryption. This notion is formalised in Section 5.3.

Signatures A signature scheme σ is a tuple of three functions $\sigma := (\mathsf{gen}, \mathsf{sig}, \mathsf{ver})$, one function to generate key pairs, one function to create signatures with a private key, and one function to verify signatures with the corresponding public verification key.

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \sigma.\mathsf{gen}()$: The key generation algorithm is a randomised algorithm that outputs a key pair $(\mathsf{sk}, \mathsf{pk})$.
- $s \leftarrow\!\!\$\ \sigma.\mathsf{sig}(\mathsf{sk}, m)$: The signature algorithm is a randomised algorithm that creates signatures s over a plaintext m for a given private key sk.
- $b \leftarrow \sigma.\mathsf{ver}(\mathsf{pk}, m, s)$: The verification algorithm is a deterministic algorithm that verifies if a signature s corresponds to a plaintext m for a given public key pk. The output value is a bit $b \in \{0,1\}$.

Moreover, a signature scheme needs to satisfy correctness. I.e.:

$$\forall(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \sigma.\mathsf{gen}()\ \forall m \in \{0,1\}^* : \Pr[\sigma.\mathsf{ver}(\mathsf{pk}, m, \sigma.\mathsf{sig}(\mathsf{sk}, m)) = 1] = 1.$$

2.4 Methodology

This thesis defines security via indistinguishability games. Namely, computational security properties are captured by the indistinguishability of a real game from an ideal game, where the latter encodes perfect security. The proofs are carried out using the State Separating Proof (SSP) methodology [BDLF+18], which is a variant of the code-based

methodology as introduced in [BR93]. In addition to traditional code-based encoding, the code of games is decomposed into collections of *packages* that can interact with each other.

Definition 1. (Package) A package is a collection of algorithms, called oracles, that may be stateful, i.e. maintain state over multiple calls, and may share state. The oracles together create one well-defined interface, the usage of oracles from other packages creates another well-defined interface.

A simple example of this is a package S to store values. To do this, S may have an interface of two oracles: STORE and GET, to store and get values in the package. Furthermore, S maintains state between calls so that a stored value can be retrieved later. The state is modified when STORE is called and accessed when GET is called.

Packages are the core components of a proof and can be composed together based on their interfaces and according to the rules laid out in [BDLF+18]. Then, an indistinguishability game can be defined as an adversaries advantage of distinguishing between interacting with one set of packages from interacting with another set of packages, both with the same interface.

Package Interfaces Every package has two well-defined interfaces: one that defines which oracles it utilises, and one that defines which oracles it provides. The former is called the in interface and the latter is called the out interface. Then, $\mathsf{in}(\mathsf{X})$ denotes the in interface of package X and $\mathsf{out}(\mathsf{X})$ denotes the out interface of package X.

As an example, the storage package S has an empty in interface and a non-empty out interface. I.e.:

$$\mathsf{in}(\mathsf{S}) := \emptyset \text{ , } \mathsf{out}(\mathsf{S}) := \{\text{STORE}, \text{GET}\}.$$

If S is used by some package X to store and get values, X will have a non-empty in interface, namely:

$$\mathsf{in}(\mathsf{X}) := \{\text{STORE}, \text{GET}\}.$$

Package Composition By composing packages, interesting behaviour emerges. Composed packages can interact with each other by calling each other's oracles. To be able to do this, the interfaces of composed packages must correspond. Packages can be composed in two ways: in sequence or in parallel. The former, following Definition 5 of [BDLF+18], is defined as:

Definition 2. (Sequential Composition) Two packages X and Y can be composed in sequence, denoted by $\mathsf{X} \circ \mathsf{Y}$, if $\mathsf{out}(\mathsf{X})$ matches $\mathsf{in}(\mathsf{Y})$. The composition results in a new package $\mathsf{Z} := \mathsf{X} \circ \mathsf{Y}$ with $\mathsf{in}(\mathsf{Z}) := \mathsf{in}(\mathsf{X})$ and $\mathsf{out}(\mathsf{Z}) := \mathsf{out}(\mathsf{Y})$.

An important observation about sequential composition, following Lemma 6 of [BDLF+18], is that it is associative. I.e.:

$$\mathsf{X} \circ (\mathsf{Y} \circ \mathsf{Z}) \overset{\text{code}}{\equiv} (\mathsf{X} \circ \mathsf{Y}) \circ \mathsf{Z}.$$

Additionally, packages can be composed in parallel. Following Definition 7 of [BDLF+18], parallel composition is defined as:

Definition 3. (Parallel Composition) Two packages X and Y can be composed in parallel, denoted by $\frac{\mathsf{X}}{\mathsf{Y}}$, if $\mathtt{out}(\mathsf{X}) \cap \mathtt{out}(\mathsf{Y}) = \emptyset$. The composition results in a new package $\mathsf{Z} := \frac{\mathsf{X}}{\mathsf{Y}}$ with $\mathtt{in}(\mathsf{Z}) := \mathtt{in}(\mathsf{X}) \cup \mathtt{in}(\mathsf{Y})$ and $\mathtt{out}(\mathsf{Z}) := \mathtt{out}(\mathsf{X}) \cup \mathtt{out}(\mathsf{Y})$.

An important observation about parallel composition, following Lemma 8 of [BDLF+18], is that it is commutative and associative. I.e.:

$$\frac{\mathsf{X}}{\mathsf{Y}} \overset{\text{code}}{\equiv} \frac{\mathsf{Y}}{\mathsf{X}}, \frac{\left(\frac{\mathsf{X}}{\mathsf{Y}}\right)}{\mathsf{Z}} \overset{\text{code}}{\equiv} \frac{\mathsf{X}}{\left(\frac{\mathsf{Y}}{\mathsf{Z}}\right)}.$$

Moreover, following Lemma 9 of [BDLF+18], it is a fact that:

$$\frac{\mathsf{X}_1}{\mathsf{X}_2} \circ \frac{\mathsf{Y}_1}{\mathsf{Y}_2} \overset{\text{code}}{\equiv} \frac{\mathsf{X}_1 \circ \mathsf{Y}_1}{\mathsf{X}_2 \circ \mathsf{Y}_2}.$$

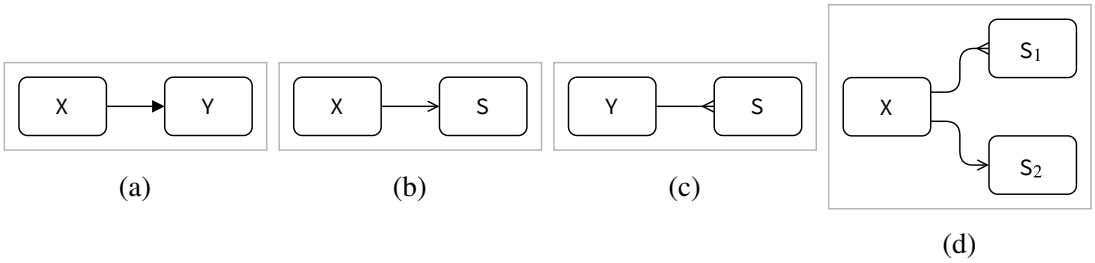
Package Indexing If a package is used multiple times in a composition, each instance of the package needs to be uniquely addressable. Following Definition 25 of [BDLF+18], this is achieved through indexing. In the present work, indexes of packages are written as a subscript on the package name. For example, if there are two instances of a package X then this is written as X_1 and X_2.

Furthermore, based on Definition 26 of [BDLF+18], the notation $\mathsf{X}[i \in \mathcal{I}]$, where $\mathcal{I} := \{1, \ldots, n\}$ is a finite non-empty set of indices, is used as a shorthand for the parallel composition of the packages X_1 to X_n.

Parameterised Packages A package definition can be parametric in some aspects. For example, a package definition may be defined in terms of a generic encryption scheme, or its behaviour can be different based on a bit value. Then, this parameterised package can be instantiated with a particular encryption scheme or a specific bit value. In the present work parameterised values are written as superscript on a package. For example, if the behaviour of package X depends on the value of a bit b then this is written as X^b.

2.4.1 Graphs

An advantage of the SSP methodology is that it can be visualised. Packages are represented as boxes with rounded corners, and oracle calls as labelled arrows between those



(a) (b) (c) (d)

Figure 2.1: Basic examples of graphs in the State Separating Proofs methodology.

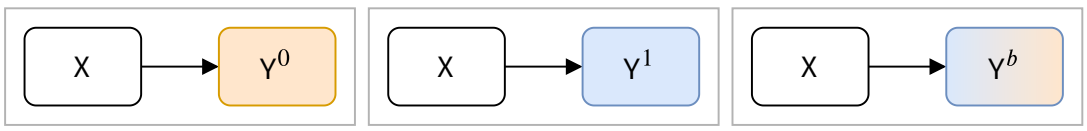
boxes. For example, Figure 2.1a shows the (sequential) composition of packages X and Y, where X has an oracle call to Y.

The present work adopts a notation where the arrowhead indicates the type of oracle call. A filled arrowhead, as seen in Figure 2.1a, indicates an oracle call that performs a complex operation, such as EVAL or ENC (the exact oracle depends on the context). An empty arrowhead, as seen in Figure 2.1b, indicates storing a value into the package with a SET or CSET query. Finally, a reverse arrowhead, as seen in Figure 2.1c, indicates retrieving a stored value from the package with a GET or CGET query, or information about a stored value.

Sequential and parallel composition can be expressed in graphs naturally through positioning. Figure 2.1a to Figure 2.1c show examples of sequential composition, as the packages are positioned along the horizontal axes. Positioning packages along the vertical axis is used for parallel composition. Figure 2.1d shows an example of the parallel composition of S_1 and S_2 in sequence with X. That is, Figure 2.1d depicts

$$\mathsf{X} \circ \frac{\mathsf{S}_1}{\mathsf{S}_2}.$$

Since SSP proofs use indistinguishability games, it is common for packages to have real and ideal behaviour. To make real and ideal package stand out in graphs a colour-coding notation, inspired by [BDLF+20], is adopted. Figure 2.2a shows an example of a composition with a real package, coloured orange, and Figure 2.2a shows an example of a composition with an ideal package, coloured blue. Lastly, Figure 2.2c shows a composition with a package that has a blue-to-orange gradient. This style is used to denote *idealisation* of the package Y with bit $b = 0$ from package Y with bit $b = 1$.



(a) (b) (c)

Figure 2.2: Real and ideal package notation in State Separating Proof graphs.

2.4.2 Proofs

Proofs in the SSP methodology largely consist of manipulating graphs according to the composition rules. A common approach is to compose the adversary $\mathcal{A}$ with a subset of packages that comprise a protocol to construct a reduction to a cryptographic assumption. To do this, the graph is cut into two parts such that one part is the cryptographic assumption. The other part, consisting of the adversary against the protocol and a composition of packages then becomes a reduction algorithm against the cryptographic assumption. See [BDLF+18] or Figure 6.3 for an example.

Advantage The present work adopts two notations to denote adversarial advantages, inspired by [BDLF+20]. First, there is a generic notation. Let $\mathcal{A}$ be an adversary, b a bit value, and $\mathsf{G}^{\alpha,b}$ (α is some parametric value of G) an indistinguishability game, then

$$\varepsilon(\mathcal{A}; \mathsf{G}^{\alpha,0}, \mathsf{G}^{\alpha,1}) := |\Pr[1 \leftarrow_{\$} \mathcal{A} \circ \mathsf{G}^{\alpha,0}] - \Pr[1 \leftarrow_{\$} \mathcal{A} \circ \mathsf{G}^{\alpha,1}]|,$$

denotes the distinguishing advantage of $\mathcal{A}$ against $\mathsf{G}^{\alpha,b}$.

Second, for common distinguishing advantages, a shorthand is adopted. Let $\mathcal{A}$ be an adversary, b a bit value, and $\mathsf{G}^{\alpha,b}$ an indistinguishability game, then

$$\varepsilon^{\alpha}_{\mathsf{G}}(\mathcal{A}) := \varepsilon(\mathcal{A}; \mathsf{G}^{\alpha,0}, \mathsf{G}^{\alpha,1}),$$

also denotes the distinguishing advantage of $\mathcal{A}$ against $\mathsf{G}^{\alpha,b}$.

Security Because MLS is a real-life protocol with concretely implemented cryptographic primitives, this thesis adopts concrete security rather than asymptotic security. This means that proofs are given in terms of advantages and exact relations between advantages, rather than security statements in terms of negligible probabilities. In practice, this means that reductions are given explicitly and that the advantage of an adversary against the MLS key schedule is related directly to the advantage of the reduction(s) against underlying cryptographic primitives. The reductions shown in this thesis are efficient given a reasonable notion of time. In particular, the discussion of time is avoided to bypass potential pitfalls such as those pointed out in [BL13, Rog06].

2.5 Key Packages

In the State Separating Proof (SSP) methodology [BDLF+18], a key package offers a composable way to store secret values so they are available to other packages while remaining unknown to the adversary. Following [BDLF+20], this is achieved through so-called handles that each refers to a single secret value. A handle should not leak any information about the secret value it refers to. The handle is made available to the

adversary who can then use it to instruct the game to perform operations. For example, to encrypt using a key the adversary does not know, the adversary provides an encryption package with the handle of the key it wants to use and the plaintext message it wants to encrypt.

In practice, the adversary may know or choose some of the secret values in a key package. For example, the adversary may know the secret key for one session but not for other sessions. Key packages keep track of which secret values are unknown to the adversary, called *honest*, and which secret values are known to the adversary, called *dishonest*.

This information about honesty is needed when packages relying on the secret values are idealised. If the ideal behaviour requires that a value is secret but the adversary knows the value, then the package needs to fall back to its real behaviour. To this end, key packages provide queries to store honest values, store dishonest values, get values, and get the honesty of values.

2.5.1 The KEY Package

A $\mathsf{KEY}^{b,\lambda}$ package, defined in Figure 2.3, stores symmetric keys and other secret values of length λ. Namely, the table K stores secret values k under a handle h. The handles for the KEY package are integers that represent the index of the value in the package.

The SET query allows for storing a secret value. In the real case ($b = 0$), SET(k) stores the provided value k, whereas in the ideal case ($b = 1$), SET(k) ignores the input and stores a randomly sampled value instead. In either case, the value is marked honest. Values in the KEY package are marked as dishonest if they are set with the CSET (*Corrupt* SET) query. This query will always store the provided value. If the value provided to SET or CSET is not of size λ the game will halt.

A value can be retrieved from a KEY package via the GET query. This query will return the secret value stored under the provided handle. If no key exists for the handle the game will halt. Similarly, a value's honesty can be retrieved from a KEY package with the HON query. If no key exists for the handle the game will halt.

Lastly, there is a CGET (*Corrupt* GET) query. This query will return the key stored under the provided handle only if the key is dishonest. This query is used in Section 5.4 and Chapter 6 to model a restricted adversarial interface.

Package Parameters	Package State
b : idealisation bit	K : **table**$[h \mapsto \mathsf{k}]$
λ : key length	H : **table**$[h \mapsto \{0,1\}]$

(a) The parameters and shared state of a KEY package.

$\mathsf{KEY}^{0,\lambda}$.SET(k)	$\mathsf{KEY}^{1,\lambda}$.SET(k)	$\mathsf{KEY}^{b,\lambda}$.CSET(k)
assert $[\lvert\mathsf{k}\rvert = \lambda]$	**assert** $[\lvert\mathsf{k}\rvert = \lambda]$	**assert** $[\lvert\mathsf{k}\rvert = \lambda]$
$h \leftarrow \lvert\mathrm{K}\rvert$	$h \leftarrow \lvert\mathrm{K}\rvert$	$h \leftarrow \lvert\mathrm{K}\rvert$
$\mathrm{K}[h] \leftarrow \mathsf{k}$	$\mathrm{K}[h] \leftarrow\!\!\!\$\ \{0,1\}^{\lambda}$	$\mathrm{K}[h] \leftarrow \mathsf{k}$
$\mathrm{H}[h] \leftarrow 1$	$\mathrm{H}[h] \leftarrow 1$	$\mathrm{H}[h] \leftarrow 0$
return h	**return** h	**return** h

$\mathsf{KEY}^{b,\lambda}$.GET(h)	$\mathsf{KEY}^{b,\lambda}$.CGET(h)	$\mathsf{KEY}^{b,\lambda}$.HON(h)
assert $[\mathrm{K}[h] \neq \perp]$	**assert** $[\mathrm{H}[h] = 0]$	**assert** $[\mathrm{H}[h] \neq \perp]$
return $\mathrm{K}[h]$	**return** $\mathrm{K}[h]$	**return** $\mathrm{H}[h]$

(b) The real and ideal KEY package query implementations.

Figure 2.3: Definition of a $\mathsf{KEY}^{b,\lambda}$ package.

2.5.2 The PKEY Package

A public-key package $\mathsf{PKEY}^{b,\zeta}$, defined in Figure 2.4, is used to store public-private key pairs for a Public Key Encryption (PKE) scheme ζ (defined in Section 2.3). Namely, the table K stores key pairs $(\mathsf{sk}, \mathsf{pk})$ under a handle h. Similar to the KEY package handles for by the PKEY package are integers that represent the index of the key pair in the package.

The SET query allows for storing a key pair. In the real case ($b = 0$), SET stores the provided key pair, whereas in the ideal case ($b = 1$), SET ignores the input and stores a randomly generated key pair instead. In either case, the value is marked as honest. Key pairs in the PKEY package are marked as dishonest if they are set with the CSET query. This query will always store the provided key pair.

A value can be retrieved from a PKEY package with the GET_SK and GET_PK queries. These queries will return the private key and public key, respectively, stored under the provided handle. If no such key pair exists the game will halt. Similarly, a key pair's honesty can be retrieved from a PKEY package with the HON query. If no key pair exists for the provided handle the game will halt.

Package Parameters	Package State
b : idealisation bit	K : **table**$[h \mapsto (\mathsf{sk}, \mathsf{pk})]$
ζ : PKE primitive	H : **table**$[h \mapsto \{0,1\}]$

(a) The parameters and shared state of a PKEY package.

$\mathsf{PKEY}^{0,\zeta}.\mathrm{SET}(\mathsf{sk}, \mathsf{pk})$	$\mathsf{PKEY}^{1,\zeta}.\mathrm{SET}(\_, \_)$	$\mathsf{PKEY}^{b,\zeta}.\mathrm{CSET}(\mathsf{sk}, \mathsf{pk})$
$h \leftarrow \lvert\mathrm{K}\rvert$	$h \leftarrow \lvert\mathrm{K}\rvert$	$h \leftarrow \lvert\mathrm{K}\rvert$
$\mathrm{K}[h] \leftarrow (\mathsf{sk}, \mathsf{pk})$	$\mathrm{K}[h] \leftarrow \zeta.\mathrm{gen}()$	$\mathrm{K}[h] \leftarrow (\mathsf{sk}, \mathsf{pk})$
$\mathrm{H}[h] \leftarrow 1$	$\mathrm{H}[h] \leftarrow 1$	$\mathrm{H}[h] \leftarrow 0$
return (pk, h)	**return** $(\mathrm{K}[h].\mathsf{pk}, h)$	**return** (pk, h)

$\mathsf{PKEY}^{b,\zeta}.\mathrm{GET\_SK}(h)$	$\mathsf{PKEY}^{b,\zeta}.\mathrm{GET\_PK}(h)$	$\mathsf{PKEY}^{b,\zeta}.\mathrm{HON}(h)$
assert $[\mathrm{K}[h] \neq \bot]$	**assert** $[\mathrm{K}[h] \neq \bot]$	**assert** $[\mathrm{H}[h] \neq \bot]$
$(\mathsf{sk}, \_) \leftarrow \mathrm{K}[h]$	$(\_, \mathsf{pk}) \leftarrow \mathrm{K}[h]$	**return** $\mathrm{H}[h]$
return sk	**return** pk	

(b) The real and ideal PKEY package query implementations.

Figure 2.4: Definition of a $\mathsf{PKEY}^{b,\zeta}$ package.

Chapter 3

Message Layer Security

The Message Layer Security (MLS) RFC [BBM+19] aims to create a continuous asynchronous group key-exchange and messaging protocol that 1) provides Forward Secrecy (FS) and Post-Compromise Security (PCS) guarantees, and 2) is efficient w.r.t. changes to the Group, even for large groups.

MLS achieves this by using a novel cryptographic primitive that can be used to obtain a common shared secret for a Group called TreeKEM [BBR18], explained in Section 3.1. In MLS, changes to a Group's TreeKEM tree are shared through Proposals and Commits, which are discussed in Section 3.2. Lastly, Section 3.3 explains the key schedule of MLS, which relies on the shared secret obtained through TreeKEM.

3.1 TreeKEM

A TreeKEM tree is a left-balanced binary tree[1] that uses a Key Derivation Function (KDF) and Key Encapsulation Method (KEM) as cryptographic primitives. The goals TreeKEM aims to achieve are:

- Distribute a secret value among Group Members.
- Allow for efficient updates to the key material of a Member and the Group.
- Allow for efficiently adding new Clients to the Group.
- Allow for efficiently removing Members from the Group.

The leaf nodes of a TreeKEM tree represent Group Members, the other nodes represent shared secrets known by the Members represented by the leaves of the subtree of the

[1] Even though it can be generalised to a tree of arbitrary arity and structure [BBR18] the present work only considers left-balanced binary tree, following the RFC.

node in question. Therefore, the secret value at the root of the tree is shared by the entire Group. Figure 3.1a illustrates which Member knows which values in the tree.

In addition, every node in the tree has associated with it a KEM key pair known to the Members represented by the leaves in the subtree. Figure 3.1b illustrates the data associated with each node. To share a secret value with some Members, a Member can encrypt that value to a node of which all intended Members know the key pair.

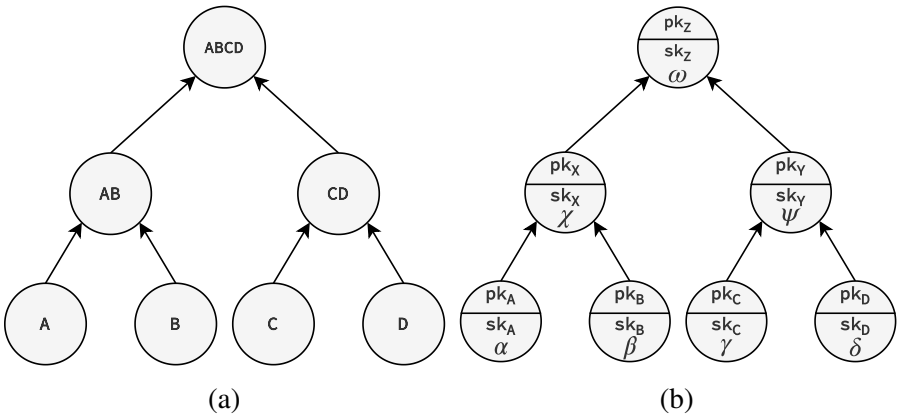


Figure 3.1: Visualisation of a TreeKEM tree. **(a)** shows which Member knows the secret data associated with which node. **(b)** shows the public data (above the line) and secret data (below the line) associated with each node.

In Figure 3.1b, the node keys $(\mathsf{sk}_A, \mathsf{pk}_A)$ of the left-most leaf node were derived from the secret value α. The value χ was deterministically derived from either value α or β. The node keys $(\mathsf{sk}_X, \mathsf{pk}_X)$ of that node were in turn derived deterministically from χ. Similarly, the value ω was derived deterministically from either value χ or ψ. The node keys $(\mathsf{sk}_Z, \mathsf{pk}_Z)$ of that node were in turn derived deterministically from ω. Which child value a parent's value was derived from depends on which Member last updated their leaf keys.

To update ones keys a new chain of secret values and keys from leaf to root can be generated and shared to the appropriate keys. This is explained in Section 3.1.1. To add a new Member to the Group, the tree structure is updated to accommodate the new Member. This is explained in Section 3.1.2. To remove a Member from the Group, the nodes of which that Member knows the keys are "blanked", i.e. marked to not be used to share new secrets. This is explained in Section 3.1.3.

3.1.1 Updating Key Material

Consider the Group represented by the tree in Figure 3.2a. The Group Member Alice (*A*) can update the Group secret by the following process. First, let *A* sample a random

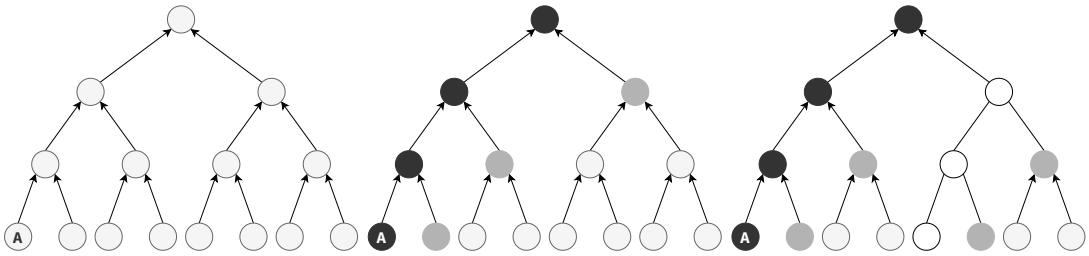


Figure 3.2: Visualisation of a Member updating their key material in a Group. **(a)** shows the tree for the Group. **(b)** shows the tree after the update, highlighting the *direct path* in black and the *co-path* in dark grey. **(c)** shows the tree after the update if there is a *blank node* in the tree.

value X and define $\texttt{path\_secret}[0] := X$. This value is used to iteratively derive secret values on the path from the leaf node of *A* to the tree root. This path is called the *direct path*.

Definition 4. (Direct Path) A list of nodes starting from a leaf node containing all ancestor nodes of the leaf node up to the root of the tree.

Figure 3.2b highlights the direct path as black nodes. Each node in the direct path has associated with it two values, which are both derived from $\texttt{path\_secret}[0]$. First, a path secret for every node along the direct path is derived as

$$\texttt{path\_secret}[i] := \kappa(\texttt{path\_secret}[i-1], \texttt{"path"}),$$

from which a *node secret* for every node is derived as

$$\texttt{node\_secret}[i] := \kappa(\texttt{path\_secret}[i], \texttt{"node"}),$$

from which a public-private key pair, referred to as node key pair, is derived as

$$(\mathsf{sk}_i, \mathsf{pk}_i) := \texttt{Derive-Key-Pair}(\texttt{node\_secret}[i]).$$

The pair associated with nodes in the direct path consist of the (new) node public key and the (new) path secret of the node. Client *A* will encrypt the path secrets in the direct path to the (other) Members of the Group in the respective subtree. However, rather than encrypting to each Member individually, Client *A* will encrypt them to the nodes in the *co-path*.

Definition 5. (Co-Path) A list of the closest non-blank descendent nodes of the nodes along with the direct path.

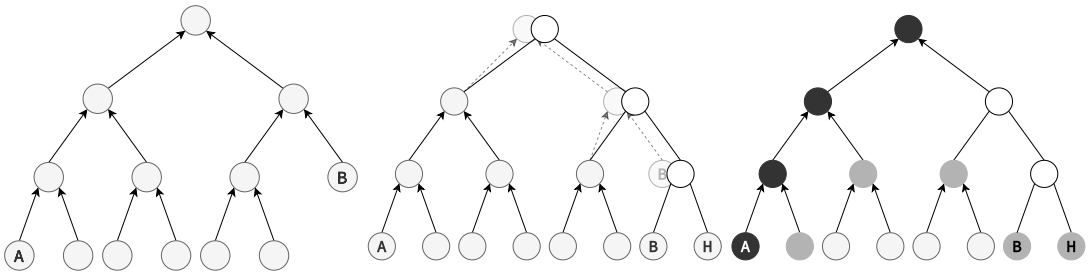
Figure 3.2b highlights the co-path as dark grey nodes. Because Client *A* received the node public keys of the nodes in the co-path in previous updates, *A* can now use the KEM to encrypt the path secrets along the new direct path using the node public keys of the co-path. I.e. for each node in the direct path, Client *A* encrypts `path_secret`$[i]$ only to the nodes in the co-path that are descendents of node i outside the direct path.

In the end, Client *A* shares the public keys and encryptions associated with the nodes in the direct path with the Group to perform the update.

3.1.2 Adding a New Member

For consistency, the RFC mandates that new Members are always added as the left-most empty leaf node in the tree. If the tree is full, the left-most leaf node is replaced with a new 3-node subtree consisting of an intermediate node, the leaf node being replaced, and the new leaf node. Furthermore, in MLS each Client makes their short-lived public-private key pairs, called ClientInitKey, publicly available. Such a key pair can be used by Members of a Group to add the Client to the Group.

Consider the Group represented by the tree in Figure 3.3a. To add a new Client Henry (*H*) to this Group, the Group Member Alice (*A*) creates a new leaf node for *H* based on one of *H*'s ClientInitKeys. Client *A* then replaces Bob's (*B*) node with a blank node and assigns the child nodes of this new node to be *B*'s node and *H*'s node. This process results in the tree in Figure 3.3b. Because of the blank node, the Members *B* and *H* do not have any shared secret between the two of them. Finally, *A* must perform an update operation. This is shown in Figure 3.3c.

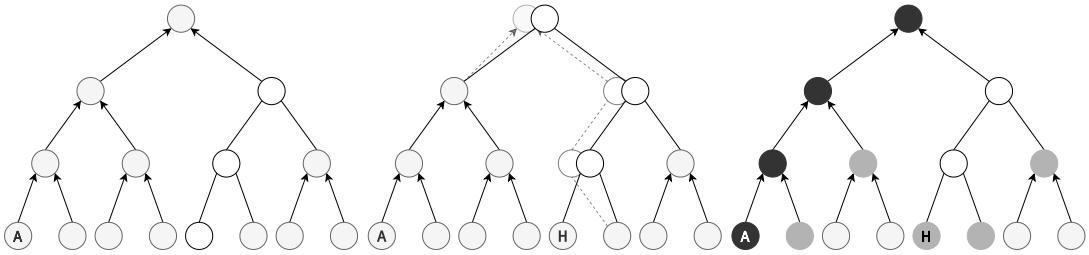


(a) (b) (c)

Figure 3.3: Visualisation of adding a new Member to a Group. **(a)** shows the tree for the original Group composition. **(b)**, overlaying the tree of **(a)**, shows the tree with *H* added to the Group, highlighting blanked nodes in white. **(c)** shows the tree for the update performed by *A*, highlighting the *direct path* in black and the *co-path* in dark grey.

If there are other blank nodes in the tree, the co-path changes as described in Section 3.1.1. If there is a blank leaf node in the tree, because a Member was previously removed, a new Member should occupy this empty spot. Figure 3.4 illustrates this

process. Here, *A* adds the new Member *H* to the group at the unoccupied leaf node at position 5. Other than that, this add functions the same as Figure 3.3.



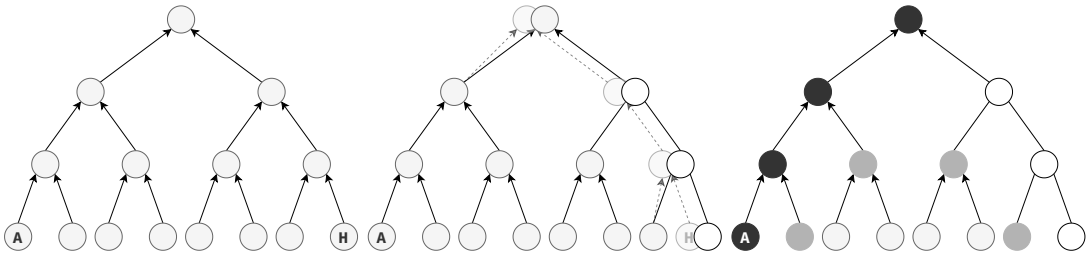
(a) (b) (c)

Figure 3.4: Visualisation of adding a new Member to a Group at an empty leaf node. **(a)** shows the tree for the original Group composition, where the leaf node at position 5 is not occupied. **(b)**, overlaying the tree of **(a)**, shows the tree with *H* added to the Group, highlighting blanked nodes in white. **(c)** shows the tree for the update performed by *A*, highlighting the *direct path* in black and the *co-path* in dark grey.

3.1.3 Removing a Member

Consider the Group represented by the tree in Figure 3.5a. 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 3.5b. As a result, a co-path that would have included those nodes now includes their non-blank child nodes.

Notice that the removal by itself does not update the Group secret. As a result *H* would still be able to decrypt message send by Group Members, breaking PCS. Therefore, *A* is required to perform an update to its leaf node. Because the tree now contains blanked nodes, the values along the new direct path must be encrypted with the keys of a larger co-path, as shown in Figure 3.5c.



(a) (b) (c)

Figure 3.5: Visualisation of removing a Member from a Group. **(a)** shows the tree for the original Group composition. **(b)**, overlaying the tree of **(a)**, shows the tree with *H* removed. **(c)** shows the 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 **(b)** and **(c)**.

3.2 Commits and Proposals

Any of the three potential changes to a TreeKEM tree (adds, removes, and updates) are shared between Group Members as Proposals. Each Proposal is sent as an individual message from one Group Member to all other Members. Upon receiving a (valid) Proposal, Group Members should cache it. Only Proposals that are valid, i.e. formatted correctly, signed by a Member, and sent in the current epoch should be cached.

A Proposal to add a new Member to the Group consists of the ClientInitKey. This is the public component of a public-private key pair that Clients make available so they can be added to Groups. A Proposal to remove a Member from the Group contains the index of the node representing the Member to be removed. Finally, a Proposal to perform an update consists of the public component of the new public-private key pair that the Member intents to start using.

Cached Proposals may be combined into a Commit by a Member of a Group. To create a commit, a Member updates the TreeKEM tree based on the Proposals it cached, following Section 3.1, and derives new keys for the Group, following Section 3.3. The Commit is distributed to other Members through a message containing the Proposals used and the public information of the direct path. The former allows recipients to update the TreeKEM tree in the same way as the committing Member, the latter allows recipients to derive the keys for the next epoch. The Commit message also contains a special Message Authentication Code (MAC) tag to verify that the newly derived Group state is correct.

In addition to a Commit, a so-called Welcome message is created. The Welcome message is intended for newly added Group Members. As such, it contains more information than a Commit message, so that new Members can compute the Group state

from scratch. Importantly, the Welcome message does not allow new Group Members to decrypt any messages from earlier epochs.

3.3 Key Schedule

The TreeKEM tree is used to distribute a common shared secret among the Members of a Group. This secret value is then used in a key schedule that evolves continuously over the lifetime of the Group with each epoch. This evolution mimics the ratcheting of the Asynchronous Ratchet Tree (ART) protocol.

In draft 8, the key schedule is used to derive 5 secret values intended for various parts of the protocol. The key schedule continuously evolves from one epoch to the next in a linear chain. To derive the keys of epoch e two values are used as inputs. First, the secret at the root of the TreeKEM tree, called the *update secret*, and second, a secret value derived in the previous epoch called the *init secret*. These two values are put into a PRF to extract proper randomness. The output value of the PRF, called the *epoch secret*, is then used to derive the 5 keys for the next epoch using a KDF. Figure 3.6 visualises this process.

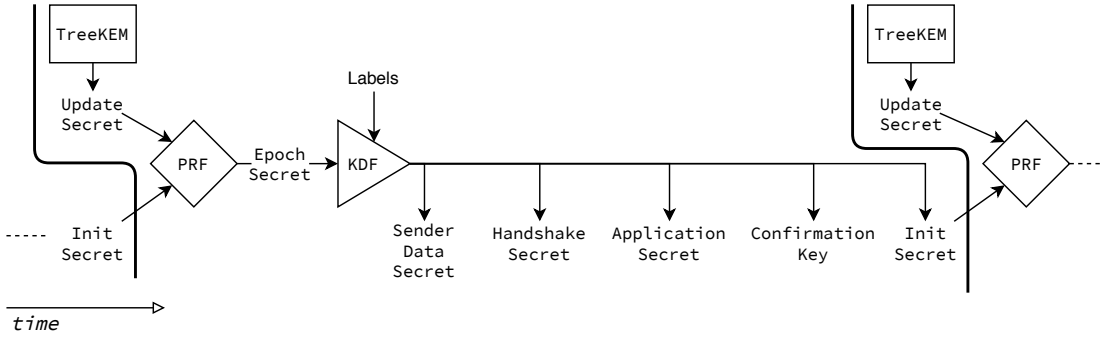


Figure 3.6: A linear graph in time of the MLS key schedule. On the far left is epoch $e - 1$, in the middle is epoch e, and on the far right is epoch $e + 1$.

The 4 secret values, besides the *init secret*, derived per epoch for a Group are called the *sender data secret*, *handshake secret*, *application secret*, and *confirmation key*. These values are used as keying material or keys for different parts of the protocol. Next, the use case of each secret is described informally.

Sender Data Secret Each message that is sent in MLS has associated with it metadata about the sender of the message, called the sender data. This metadata is protected by encrypting it under a key derived from the sender data secret. To goal of this encryption is to protect the metadata from third-parties that are not in the Group. The sender data

key is derived as

$$\mathsf{sender\_data\_key} := \kappa(\mathsf{sender\_data\_secret}, \texttt{"sd key"}).$$

Handshake Secret When a message concerns the Group's TreeKEM tree (a Proposal or a Commit) it is called a handshake message. Such messages are encrypted using a per-Member key and nonce derived from the handshake secret. The key is the same for all handshake message and is computed as

$$\mathsf{handshake\_key} := \kappa(\mathsf{handshake\_secret} \| i_{\text{sender}}, \texttt{"hs key"}),$$

where i_{sender} is the index of the leaf node of the sender of the message. The nonce is different for every handshake message. The first nonce is computed as

$$\mathsf{handshake\_nonce}_0 := \kappa(\mathsf{handshake\_secret} \| i_{\text{sender}}, \texttt{"hs nonce"}),$$

where i_{sender} is the index of the leaf node of the sender of the message. Subsequent nonces are computed as

$$\mathsf{handshake\_nonce}_g := \mathsf{handshake\_nonce}_0 \oplus g,$$

where g (short for *generation*) is the (big endian encoded) index of the handshake message for a sender.

Confirmation Key Each time a Group advances to the next epoch, a MAC tag is included so Members can verify that they derived the correct Group state. The key used to compute and verify this MAC tag is the confirmation key. It is used directly as it is derived from the init secret and update secret. Since the key is used only once per epoch (or a few times in case a Commit fails), it is not updated within an epoch.

Application Secret Lastly, there are messages in MLS with the purpose of human communication between Group Members. Communication messages are encrypted with a separate set of keys. As with handshake messages, these keys are derived per Member and are ratcheted with each message to provide FS within an epoch. However, the ratcheting for communication message is more complicated than the ratcheting for handshake messages. Since communication message are outside the scope of the present work their derivation logic will not be covered here.

Chapter 4

Model

In this chapter the informal description of MLS from the RFC [BBM+19] and Chapter 3 is transformed into a formal definition. First, Section 4.1 specifies the adversarial model. Next, Section 4.2 gives the syntax of a continuous asynchronous group key agreement (CAGKA) protocol. Finally, Section 4.3 gives the implementation of MLS as a CAGKA protocol.

The present work does not provide a security proof of MLS as a CAGKA in its entirety. Instead, only security proofs for TreeKEM and the key-exchange of MLS are provided. Section 8.2 discusses how the results of the present work can be used to perform a complete security analysis in the future.

4.1 Adversarial Model

The adversarial model for this thesis is one where the adversary is *active* and *static* (or *non-adaptive*). An active adversary is in control of the actions taken by the participants in a protocol as well as the network. A *static* adversary is one which is required to announce all corruptions upfront and cannot corrupt dynamically.

4.2 Group Messaging Protocol Syntax

This section defines the syntax of a continuous asynchronous group key agreement (CAGKA) protocol Π that is used as the basis for the security analysis in this thesis, although it is not covered fully. $\mathcal{C}$ is used to denote a Client or a Member, $\mathcal{G}$ to denote a Group, $\mathcal{S}$ to denote *sessions*, and e to denote an epoch. A *session* in a CAGKA protocol is the state of a Group from the perspective of a single Member.

A CAGKA protocol Π is a tuple of ten algorithms that are executed in the context of one Client $\mathcal{C}$ and optionally one session $\mathcal{S}$. If the arguments provided to the algorithm are incorrect the return value must be $\diamond$. The algorithms are:

- $\mathcal{C} \leftarrow\!\!\$\ \Pi.\text{NewClient}()$: is a randomised algorithm that initialises a new Client. The return value of this algorithm is the newly created Client object $\mathcal{C}$.

- $\mathcal{S} \leftarrow\!\!\$\ \Pi.\text{NewGroup}(\mathcal{C})$: is a randomised algorithm that initialises a new Group $\mathcal{G}$ with Client $\mathcal{C}$ as only Member.[1] The return value of this algorithm is a session $\mathcal{S}$ representing the initial state of $\mathcal{G}$ from the perspective of $\mathcal{C}$.

- $(\mathcal{C}', cik) \leftarrow\!\!\$\ \Pi.\text{CreatePreSharedKey}(\mathcal{C})$: is a randomised algorithm that generates a new short-lived key pair for the Client $\mathcal{C}$ and a corresponding "certificate" by $\mathcal{C}$. The public component of this key pair can be used by other Clients to add $\mathcal{C}$ to a Group. The return value for this algorithm is the certificate cik and an updated Client object $\mathcal{C}'$.[2]

- $(\mathcal{S}', p) \leftarrow\!\!\$\ \Pi.\text{ProposeAdd}(\mathcal{S}, cik)$: is a possibly randomised algorithm that creates a Proposal to add a Client $\mathcal{C}_s$, that owns cik, to the Group $\mathcal{G}$ represented by $\mathcal{S}$. The return value for this algorithm is a Proposal message p intended for the Members of $\mathcal{G}$ and an updated session $\mathcal{S}'$ for $\mathcal{C}$.

- $(\mathcal{S}', p) \leftarrow\!\!\$\ \Pi.\text{ProposeRemove}(\mathcal{S}, \mathsf{pk})$: is a possibly randomised algorithm that creates a Proposal to remove a Member $\mathcal{C}_s$, identified by pk, from the Group $\mathcal{G}$ represented by $\mathcal{S}$. The return value for this algorithm is a Proposal message p intended for the Members of $\mathcal{G}$ and an updated session $\mathcal{S}'$ for $\mathcal{C}$.

- $(\mathcal{S}', p) \leftarrow\!\!\$\ \Pi.\text{ProposeUpdate}(\mathcal{S})$: is a possibly randomised algorithm that creates a Proposal to update its keying material for the Group $\mathcal{G}$ represented by $\mathcal{S}$. The return value for this algorithm is a Proposal message p intended for the Members of $\mathcal{G}$ and an updated session $\mathcal{S}'$ for $\mathcal{C}$.

- $\mathcal{S}' \leftarrow \Pi.\text{ReceiveProposal}(\mathcal{S}, p)$: is a deterministic algorithm that updates the session when a Proposal is received. The return value for this algorithm is an updated session $\mathcal{S}'$ for $\mathcal{C}$.

- $(\mathcal{S}', c, w) \leftarrow\!\!\$\ \Pi.\text{Commit}(\mathcal{S})$: is a possibly randomised algorithm that creates a Commit for the session $\mathcal{S}$. The return value of this algorithm is a Commit c intended for the Members of $\mathcal{G}$, a Welcome message w for all Clients added to $\mathcal{G}$ and an updated session $\mathcal{S}'$ for $\mathcal{C}$.

- $\mathcal{S}_{e+1} \leftarrow \Pi.\text{ReceiveCommit}(\mathcal{S}_e, c)$: is a deterministic algorithm that decodes, validates, and applies a Commit c to the session $\mathcal{S}_e$. The return value of this algorithm is an updated session $\mathcal{S}_{e+1}$ for $\mathcal{C}$. If $\mathcal{C}$ was removed from $\mathcal{G}$ the return value must be $\perp$.

[1] In the present work, group creation consists only of the initialisation of a one-member group. In the MLS RFC adding the initial group members is part of the group initialisation.

[2] In MLS the certificate is called a "ClientInitKey".

- $\mathcal{S} \leftarrow \Pi$.ReceiveWelcome$(\mathcal{C}, w)$: is a deterministic algorithm that decodes and validates a Welcome message w for a Client $\mathcal{C}$. The return value of this algorithm is a session $\mathcal{S}$ representing the Group state from the perspective of $\mathcal{C}$ of the Group to which it was added.

Remark The ability of outsiders to propose changes to a Group $\mathcal{G}$, present in the MLS RFC, is missing from this CAGKA definition. Since the propose algorithms require a session $\mathcal{S}$, a Client $\mathcal{C} \notin \mathcal{G}$ cannot create a Proposal for group $\mathcal{G}$. For completeness w.r.t. MLS, the CAGKA definition should be extended to allow outsiders to propose Group changes. However, this is beyond the scope of the present work.

4.3 MLS Protocol Implementation

This section gives the implementation of MLS as a CAGKA protocol. First, Section 4.3.1 defines the data stored in Client objects $\mathcal{C}$ and session objects $\mathcal{S}$. Next, Section 4.3.2 provides the pseudocode of MLS as a CAGKA protocol. This section only provides the pseudocode of algorithms that are relevant to the proof in Chapter 6 and Chapter 7. The pseudocode for the remaining algorithms can be found in Appendix C.

Remark The implementation presented in this thesis omits 1) values that indicate the protocol version, as this thesis is only concerned with draft 8, and 2) values that indicate cypher suites, assuming for simplicity that there is only a single cypher suite in use.

4.3.1 Stateful Objects

For MLS, a Client object $\mathcal{C}$ holds the following state:

- $\mathcal{C}$.id: The identifier of the Client $\mathcal{C}$.
- $\mathcal{C}.\mathsf{sk}_{\mathsf{ll}}$: The long-lived private or signing key of Client $\mathcal{C}$.
- $\mathcal{C}.\mathsf{pk}_{\mathsf{ll}}$: The long-lived public or verification key of Client $\mathcal{C}$.
- $\mathcal{C}.cc$: The credentials of the long-lived public-private key of Client $\mathcal{C}$.
- $\mathcal{C}.psk$: The ClientInitKeys (or pre-shared keys) of Client $\mathcal{C}$.

And a session object $\mathcal{S}$ holds the following state:

- $\mathcal{S}$.id: The identifier of the Group represented by $\mathcal{S}$.
- $\mathcal{S}$.epoch: The index of the epoch in the Group represented by $\mathcal{S}$.
- $\mathcal{S}$.th: The hash of the root of the TreeKEM tree in the Group represented by $\mathcal{S}$.

- $\mathcal{S}$.cth: The *confirmed transcript hash* of the Group represented by $\mathcal{S}$.
- $\mathcal{S}$.context: An object representing the `GroupContext` of the Group represented by $\mathcal{S}$. This consists of $\mathcal{S}$.id, $\mathcal{S}$.epoch, $\mathcal{S}$.th, and $\mathcal{S}$.cth.
- $\mathcal{S}$.rt: The TreeKEM tree of the Group represented by $\mathcal{S}$.
- $\mathcal{S}$.ith: The *interim transcript hash* of the Group represented by $\mathcal{S}$.
- $\mathcal{S}$.proposals: The cached Proposals in the Group represented by $\mathcal{S}$.
- $\mathcal{S}$.keys: The secret keys in the Group represented by $\mathcal{S}$ for a single epoch, i.e. the *epoch secret*, *application secret*, *init secret*, *sender data key*, *handshake key*, and *confirmation key* for the next epoch. As well as the key pairs of the Client $\mathcal{C}$ for that Group.
- $\mathcal{S}.g_{hk}$: The generation of handshake messages for the current epoch of the Group represented by $\mathcal{S}$.

4.3.2 Implementation

Algorithm 1, Algorithm 2, and Algorithm 3 show the implementation of the CAGKA algorithms Π.Commit, Π.ReceiveCommit, and Π.ReceiveWelcome for MLS. Algorithm 7 to Algorithm 6 show the implementation of helper functions related to the direct path and key schedule. Other helper functions, denoted by Π^*, can be found in Appendix C. Comments are employed to indicate the structures defined in the RFC that certain variables represented.

Remark In draft 8, the order in which Proposals should be applied is not explicit. Therefore, the assumed order is 1) updates, 2) removals, 3) adds. This approach has also been adopted in the changes leading up to draft 9.

Also, in draft 8 the description of how to create a commit is incomplete. Therefore, the description present in git commit `de04e9a` leading up to draft 9 is used in the present work.

Algorithm 1 Π.Commit($\mathcal{S}$)

$(\mathcal{S}, \_) \leftarrow \Pi$.ProposeUpdate($\mathcal{S}$)
If $[\mathcal{S} = \diamond]$ **return** $(\diamond, \diamond, \diamond)$

$\{\text{gid}, e, rt_e, cth_e, ith_e, \ldots\} \leftarrow \mathcal{S}$
$\{\mathsf{sk}_{\text{ll}}, \mathsf{sk}_{\text{sl}}, \mathsf{pk}_{\text{sl}}, \mathtt{is}_e, \ldots\} \leftarrow \mathcal{S}$.keys

$ps \leftarrow \mathcal{S}$.proposals
pids $\leftarrow \Pi^*$.ProposalsToList(ps)

$rt_{e+1} \leftarrow \Pi^*$.ApplyProposals($rt_e, ps,$ pids)
$gc_* \leftarrow \{\text{gid}, (e+1), 0, cth_e\}$ ▷ `GroupContext`
$path \leftarrow \Pi^*$.CreateDirectPath($rt_{e+1}, \mathsf{sk}_{\text{sl}}, gc_*$)
$c \leftarrow \{\text{pids}, path\}$ ▷ `Commit`

$i_{\text{sender}} \leftarrow \Pi^*$.GetSenderIndex($rt_{e+1}, \mathsf{pk}_{\text{sl}}$)
$cc \leftarrow \{\text{gid}, e, i_{\text{sender}}, 3, ps, c\}$ ▷ `CommitContent`
$cth_{e+1} \leftarrow \text{H}(ith_e \| cc)$

$(\_, th_{e+1}) \leftarrow \Pi^*$.ComputeTreeRoot($rt_{e+1}, gc_*, \mathsf{sk}_{\text{sl}}, path$)
$gc_{e+1} \leftarrow \{\text{gid}, (e+1), th_{e+1}, cth_{e+1}\}$ ▷ `GroupContext`
$(\mathsf{ck}_{e+1}, \ldots) \leftarrow \Pi^*$.ComputeGroupKeys($\mathtt{is}, th_{e+1}, gc_{e+1}$)
$conf \leftarrow \mu$.tag($\mathsf{ck}_{e+1}, cth_{e+1}$)

$content \leftarrow \{ps, c, conf\}$
$(\mathcal{S}, ct) \leftarrow \Pi^*$.EncodeMessage($\mathcal{S}, 3, content$)

$cad \leftarrow \{ct.s, conf\}$ ▷ `CommitAuthData`
$ith_{e+1} \leftarrow \text{H}(cth_{e+1} \| cad)$

$si \leftarrow \{\text{gid}, (e+1), rt_{e+1}, cth_{e+1}, ith_{e+1}, path, conf, i_{\text{sender}}\}$
$s \leftarrow \sigma$.sig($\mathsf{sk}_{\text{ll}}, si$)
$gi \leftarrow si \cup \{s\}$ ▷ `GroupInfo`

$n \leftarrow\!\!\$\ \{0,1\}^8$
$kps \leftarrow [\{\text{H}(\mathsf{pk}_{\text{sl}}), \zeta.\text{enc}(\mathsf{pk}_{\text{sl}}, \mathtt{is}_e)\}$ **for** $\{\mathsf{pk}_{\text{sl}}\} \in Added(\text{pids})]$

$gi_{\text{enc}} \leftarrow\!\!\$\ \xi$.enc($\mathsf{k}, n, gi, 0$)
$w \leftarrow \{kps, gi_{\text{enc}}\}$ ▷ `Welcome`

return $(\mathcal{S}, ct, w)$

Algorithm 2 Π.ReceiveCommit$(\mathcal{S}_e, ct)$

$\{\text{gid}, e, rt_e, cth_e, ith_e, \ldots\} \leftarrow \mathcal{S}_e$
$\{\mathsf{sk}_{\text{sl}}, \mathsf{pk}_{\text{sl}}, \mathsf{sk}_{\text{ll}}, \mathsf{pk}_{\text{ll}}, \mathtt{is}_e, \ldots\} \leftarrow \mathcal{S}_e$.keys

$pt \leftarrow \Pi^*$.DecodeMessage$(\mathcal{S}_e, ct)$
if $[pt = \diamond]$ **return** $\diamond$

$\{\text{gid}', e', i_{\text{sender}}, ps, c, conf, \ldots\} \leftarrow pt$
if $[\text{gid} \neq \text{gid}'] \vee [e \neq e']$ **return** $\diamond$

$\{\text{pids}, path\} \leftarrow c$
$rt_{e+1} \leftarrow \Pi^*$.ApplyProposals$(rt_e, ps, \text{pids})$

$cc \leftarrow \{\text{gid}, e, i_{\text{sender}}, 3, c\}$ ▷ `CommitContent`
$cth_{e+1} \leftarrow \text{H}(ith_e \| cc)$

$gc_* \leftarrow \{\text{gid}, (e+1), 0, cth_e\}$
for $\mathsf{sk}_{\text{sl}} \in \mathcal{S}_e$.keys **do**
  $(\mathcal{S}', th_{e+1}) \leftarrow \Pi^*$.ComputeTreeRoot$(rt_{e+1}, gc_*, \mathsf{sk}_{\text{sl}}, path)$
  if $[th_{e+1} \neq \diamond]$ **then**
    $\mathcal{S} \leftarrow \mathcal{S}'$
    break
  end if
end for
if $[th_{e+1} = \perp]$ **return** $\perp$

$gc_{e+1} \leftarrow \{\text{gid}, (e+1), th_{e+1}, cth_e\}$ ▷ `GroupContext`
$(\mathsf{ck}_{e+1}, \mathtt{is}_{e+1}, \ldots) \leftarrow \Pi^*$.ComputeGroupKeys$(\mathtt{is}_e, th_{e+1}, gc_{e+1})$
if $\neg\mu$.ver$(\mathsf{ck}_{e+1}, conf, cth_{e+1})$ **return** $\diamond$

$cad \leftarrow \{ct.s, conf\}$ ▷ `CommitAuthData`
$ith_{e+1} \leftarrow \text{H}(cth_{e+1} \| cad)$

$\mathcal{S}_{e+1} \leftarrow \{gc_{e+1}, rt_{e+1}, ith_{e+1}\}$
$\mathcal{S}_{e+1}$.keys $\leftarrow \{\mathsf{sk}_{\text{sl}}, \mathsf{pk}_{\text{sl}}, \mathsf{sk}_{\text{ll}}, \mathsf{pk}_{\text{ll}}\} \cup \Pi^*$.ComputeGroupKeys$(\mathtt{is}_e, th_{e+1}, gc_{e+1})$
return $\mathcal{S}_{e+1}$

Algorithm 3 Π.ReceiveWelcome($\mathcal{C}, w$)

$\{kps, gi_{\text{enc}}\} \leftarrow w$
$\{\text{H}(\mathsf{pk}_{\text{sl}}), \mathsf{k}_{\text{enc}}\} \leftarrow$ key corresponding to a sk_{sl} from $\mathcal{C}$ in kps
if not found **return** $\diamond$

$\mathsf{k} \leftarrow \zeta$.dec($\mathsf{sk}_{\text{sl}}, \mathsf{k}_{\text{enc}}$)
$gi \leftarrow \xi$.dec($\mathsf{k}, n, gi_{\text{enc}}, 0$)
if $[gi = \perp]$ **return** $\diamond$

$\{\text{gid}, e, th, rt, cth, ith, path, conf, i_{\text{sender}}\} \leftarrow gi$
$si \leftarrow gi \setminus \{s\}$
$\mathsf{pk}_{\text{ll}} \leftarrow$ verification key at index i_{sender} in rt
if $\neg\sigma$.ver($\mathsf{pk}_{\text{ll}}, si, \sigma$) **return** $\diamond$

$gc_* \leftarrow \{\text{gid}, (e+1), 0, cth\}$
$th \leftarrow \Pi^*$.ComputeTreeRoot($rt_{e+1}, gc_*, \mathsf{sk}_{\text{sl}}, path$)
$gc \leftarrow \{\text{gid}, e, th, cth\}$ ▷ `GroupContext`

$(\mathsf{ck}, \ldots) \leftarrow \Pi^*$.ComputeGroupKeys($\mathsf{k}, th, gc$)
if $\neg\mu$.ver($\mathsf{ck}, conf, cth$) **return** $\diamond$

$\mathcal{S} \leftarrow \{gc, rt, ith\}$
$\mathcal{S}$.keys $\leftarrow \{\mathsf{sk}_{\text{sl}}, \mathsf{pk}_{\text{sl}}, \mathcal{C}.\mathsf{sk}_{\text{ll}}, \mathcal{C}.\mathsf{pk}_{\text{ll}}\} \cup \Pi^*$.ComputeGroupKeys($\mathsf{k}, th, gc$)
return $\mathcal{S}$

4.3.3 Advancing the Key Schedule

Algorithm 4 Π^*.ExpandLabel($secret, label, context, length, gc$)

$hkdflabel \leftarrow \{\text{H}(gc), length, label, context\}$ ▷ `HkdfLabel`
return $\kappa(secret, hkdflabel, length)$

Algorithm 5 Π^*.DeriveSecret($secret, label, gc$)

return Π^*.ExpandLabel($secret, label,$ `""`$, \Pi^*$.HashLen$, gc$)

Algorithm 6 Π^*.ComputeGroupKeys($\mathsf{is}_e, \mathsf{us}_{e+1}, gc$)

$\mathsf{es} \leftarrow \varphi(\mathsf{us}_{e+1}, \mathsf{is}_e)$
$sds_{e+1} \leftarrow \Pi^*$.DeriveSecret($\mathsf{es}$, "sender data", gc)
$\mathsf{hk}_{e+1} \leftarrow \Pi^*$.DeriveSecret($\mathsf{es}$, "handshake", gc)
$\mathsf{as}_{e+1} \leftarrow \Pi^*$.DeriveSecret($\mathsf{es}$, "app", gc)
$\mathsf{ck}_{e+1} \leftarrow \Pi^*$.DeriveSecret($\mathsf{es}$, "confirm", gc)
$\mathsf{is}_{e+1} \leftarrow \Pi^*$.DeriveSecret($\mathsf{es}$, "init", gc)
return $\{sds_{e+1}, \mathsf{hk}_{e+1}, \mathsf{as}_{e+1}, \mathsf{ck}_{e+1}, \mathsf{is}_{e+1}\}$

4.3.4 Direct Path

Algorithm 7 Π^*.CreateDirectPath($rt, \mathsf{sk}_{\mathsf{sl}}, gc$)

$node \leftarrow$ leaf of $\mathsf{sk}_{\mathsf{sl}}$ in rt
$ps[0] \leftarrow\!\!\$\ \{0,1\}^{32}$
$path \leftarrow [\,]$
while parent($node$) **do**
  $node \leftarrow$ parent($node$)
  $ps[i] \leftarrow \Pi^*$.ExpandLabel($ps[i-1]$, "path", "", Π^*.HashLen, gc)
  $ns[i] \leftarrow \Pi^*$.ExpandLabel($ps[i]$, "node", "", Π^*.HashLen, gc)
  $(\_, \mathsf{pk}_i) \leftarrow$ Derive-Key-Pair($ns[i]$)
  $ks \leftarrow \emptyset$
  for $node' \in$ copath($node$) **do**
    $ks[node'] \leftarrow\!\!\$\ \zeta$.enc($node'$.pk, $ps[i]$)
  end for
  $path \leftarrow\!\!/\!\!/\ (\mathsf{pk}_i, ks)$
end while
return $path$

Algorithm 8 Π^*.ComputeTreeRoot$(rt, gc, \mathsf{sk}_{\text{sl}}, path)$

$\{gc, \ldots\} \leftarrow \mathcal{S}$
for $node \in path$ **do**
  $(\mathsf{pk}_i, ks) \leftarrow node$
  $i \leftarrow$ position of $node$ in $\mathcal{S}$.rt
  $\mathcal{S}$.rt$[i]$.pk $\leftarrow \mathsf{pk}_i$
  if $[ps_i \neq \perp]$ **then**
    $ps_i \leftarrow \Pi^*$.ExpandLabel$(ps_i, \texttt{"path"}, \texttt{""}, \Pi^*.\text{HashLen}, gc)$
    $ns_i \leftarrow \Pi^*$.ExpandLabel$(ps_i, \texttt{"node"}, \texttt{""}, \Pi^*.\text{HashLen}, gc)$
    $(\mathsf{sk}_i, \_) \leftarrow \texttt{Derive-Key-Pair}(ns_i)$
    $\mathcal{S}$.rt$[i]$.sk $\leftarrow \mathsf{sk}_i$
  else if i am a descendent of $node$ **then**
    $\{\mathsf{sk}_{\text{sl}}, \ldots\} \leftarrow \mathcal{S}$.keys
    $leaf \leftarrow$ leaf of sk_{sl} in $\mathcal{S}$.rt
    for $k \in ks$ **do**
      $node' \leftarrow leaf$
      while parent$(node')$ **do**
        $ps_i \leftarrow\!\!\$\ \zeta$.dec$(node'.\mathsf{sk}, k)$
        if $[ps_i \neq \perp]$ **break**
        $node' \leftarrow$ parent$(node')$
      end while
      if $[ps_i \neq \perp]$ **break**
    end for
    if $[ps_i = \perp]$ **return** $(\diamond, \diamond)$
  end if
end for
return $(\mathcal{S}, \Pi^*.\text{ExpandLabel}(ps_i, \texttt{"path"}, \texttt{""}, \Pi^*.\text{HashLen}, gc))$

Chapter 5

Cryptographic Assumptions

This chapter defines the cryptographic assumptions used to prove the security of TreeKEM and the key schedule of MLS. The assumptions are w.r.t. a Double Pseudorandom Function (Section 5.1), a Key Derivation Function (Section 5.2), a Public Key Encryption scheme (Section 5.3 and Section 5.4).

5.1 Dual Pseudorandom Function

A Dual Pseudorandom Function (DPRF) is a PRF that computes one output value from exactly two input values. The security property of this primitive is that the output value is honest if at least one of the input values is honest. This section defines the DPRF assumption that will be used in the present work.

Let φ be a PRF with the syntax defined in Section 2.3, $\mathcal{A}$ an adversary, and $\text{GDPRF}_{\varphi}^{b,\lambda_{s1},\lambda_{s2},\lambda_t}$ the indistinguishability game defined in Figure 5.1. Then, the DPRF advantage is defined as

$$\varepsilon_{\text{DPRF}}^{\varphi}(\mathcal{A}) := |\text{Pr}[1 \leftarrow\!\!\$\ \mathcal{A} \circ \text{GDPRF}_{\varphi}^{0,\lambda_{s1},\lambda_{s2},\lambda_t}] - \text{Pr}[1 \leftarrow\!\!\$\ \mathcal{A} \circ \text{GDPRF}_{\varphi}^{1,\lambda_{s1},\lambda_{s2},\lambda_t}]|,$$

where λ_{s1}, λ_{s2}, and λ_t are the source and target key sizes respectively.

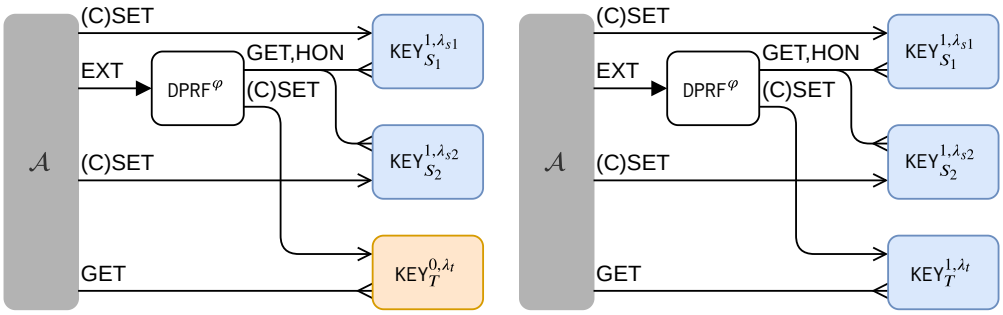


Figure 5.1: The $\text{GDPRF}^{b,\lambda_{s1},\lambda_{s2},\lambda_t}_{\varphi}$ game definition. The DPRF package is defined in Figure 5.2 and (C)SET $:=$ (SET, CSET).

Package Parameters	Package State
φ : PRF primitive	H : **table**$[(\mathsf{k},\mathsf{k}) \mapsto h]$

(a) The parameters and shared state of a DPRF package.

$\mathsf{DPRF}^{\varphi}.\text{EXT}(h_{s1}, h_{s2})$

$s_1 \leftarrow \mathsf{KEY}_{S_1}.\text{GET}(h_{s1})$
$s_2 \leftarrow \mathsf{KEY}_{S_2}.\text{GET}(h_{s2})$
if $[\text{H}[(s_1, s_2)] = \perp]$ **then**
  $t \leftarrow \varphi(s_1, s_2)$
  if $\mathsf{KEY}_{S_1}.\text{HON}(h_{s1}) \vee \mathsf{KEY}_{S_2}.\text{HON}(h_{s2})$ **then**
    $\text{H}[(s_1, s_2)] \leftarrow^{\$} \mathsf{KEY}_{T}.\text{SET}(t)$
  else
    $\text{H}[(s_1, s_2)] \leftarrow \mathsf{KEY}_{T}.\text{CSET}(t)$
  end if
end if
return $\text{H}[(s_1, s_2)]$

(b) The DPRF package query implementation.

Figure 5.2: Definition of a DPRF^{φ} package. The package state is used to guarantee idempotency between calls for identical values. If omitted, an ideal $\mathsf{KEY}_T.\text{SET}(\cdot)$ will sample two independent random values for the same (honest) input, which would allow $\mathcal{A}$ to trivially distinguish $\text{GDPRF}^{b,\lambda_{s1},\lambda_{s2},\lambda_t}_{\varphi}$.

5.2 Key Derivation Function

A Key Derivation Function (KDF) is a function that takes as input a value and a label and extracts a pseudorandom value. The goal behind this construction is that several values can be extracted from the same input value for different labels such that outputs are independent and pseudorandom. This section defines the KDF assumption that will be used in the present work.

Let κ be a KDF with the syntax defined in Section 2.3, $\mathcal{A}$ an adversary, and $\text{GKDF}_{\kappa}^{b,n,\lambda_s,\lambda_t}$ the indistinguishability game defined in Figure 5.3. Then, for any n, the KDF advantage is defined as

$$\varepsilon_{\text{KDF}}^{\kappa}(\mathcal{A}) := |\Pr[1 \leftarrow\!\$\ \mathcal{A} \circ \text{GKDF}_{\kappa}^{0,n,\lambda_s,\lambda_t}] - \Pr[1 \leftarrow\!\$\ \mathcal{A} \circ \text{GKDF}_{\kappa}^{1,n,\lambda_s,\lambda_t}]|,$$

where λ_s and λ_t are the source and target key sizes respectively.

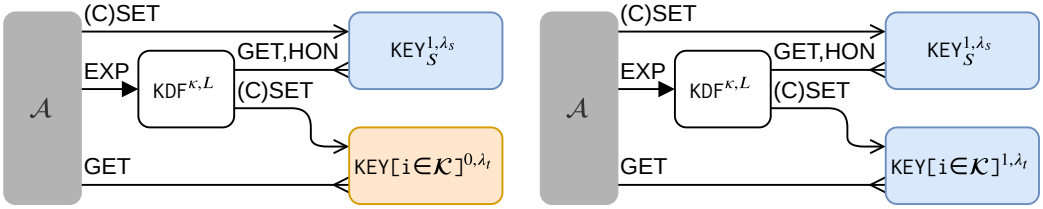


Figure 5.3: The $\text{GKDF}_{\kappa}^{b,n,\lambda_s,\lambda_t}$ game definition, where $\mathcal{K} := \{1,\ldots,n\}$ and $L := \{L_1,\ldots,L_n\}$ (a set of n labels). The KDF package is defined in Figure 5.4.

Package Parameters	Package State
κ : KDF primitive L : set of labels	H : **table**$[h \mapsto [h, \ldots]]$

(a) The parameters and shared state of a KDF package.

$\mathsf{KDF}^{\kappa,L}.\text{EXP}(h_s)$

$s \leftarrow \mathsf{KEY}_S.\text{GET}(h_s)$
if $[\text{H}[s] = \perp]$ **then**
  $\text{H}[s] \leftarrow [\,]$
  for $L_i \in L$ **do**
    $t_i \leftarrow \kappa(s, L_i)$
    if $\mathsf{KEY}_S.\text{HON}(h_s)$ **then**
      $h_i \leftarrow\!\!\$\ \mathsf{KEY}_i.\text{SET}(t_i)$
    else
      $h_i \leftarrow \mathsf{KEY}_i.\text{CSET}(t_i)$
    end if
    $\text{H}[s] \leftarrow\!\!/\!/\ h_i$
  end for
end if
return $\text{H}[s]$

(b) The KDF package query implementation.

Figure 5.4: Definition of a $\mathsf{KDF}^{\kappa,L}$ package. The package state is used to guarantee idempotency between calls for identical values. If omitted, an ideal $\mathsf{KEY}_i.\text{SET}(\cdot)$ will sample two independent random values for the same (honest) input, which would allow $\mathcal{A}$ to trivially distinguish $\text{GKDF}_{\kappa}^{b,n,\lambda_s,\lambda_t}$.

5.2.1 KDF with CGET

In Chapter 6, an alternative but equivalent assumption on a KDF is required where the adversary has access to an additional query to get dishonest keys from KEY_S. This additional CGET query does not provide additional information to the adversary since it knows the dishonest keys by definition. However, in Chapter 6, different parts of the system will make the CSET and CGET query, see Figure 6.2, one part chooses dishonest keys whereas the other part of the system merely reads the keys. The additional CGET query allows for expressing this difference.

Let κ be a KDF with the syntax defined in Section 2.3, $\mathcal{A}$ an adversary, and

$\text{GKDF2}_\kappa^{b,n,\lambda_s,\lambda_t}$ the indistinguishability game defined in Figure 5.5. Then, for any n, the KDF with CGET advantage is defined as

$$\varepsilon^\kappa_{\text{KDF2}}(\mathcal{A}) := |\Pr[1 \leftarrow\!\$\ \text{GKDF2}_\kappa^{0,n,\lambda_s,\lambda_t}] - \Pr[1 \leftarrow\!\$\ \text{GKDF2}_\kappa^{1,n,\lambda_s,\lambda_t}]|,$$

where λ_s and λ_t are the source and target key sizes respectively. Lemma B.0.1 shows the equivalence between the two notions of KDF security.

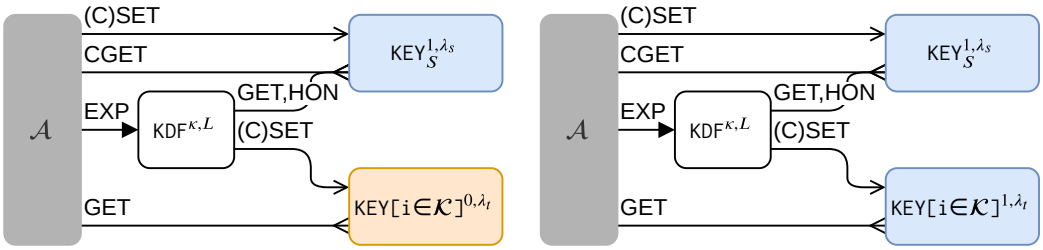


Figure 5.5: The $\text{GKDF2}_\kappa^{b,n,\lambda_s,\lambda_t}$ game definition, where $\mathcal{K} := \{1,\ldots,n\}$ and $L := \{L_1,\ldots,L_n\}$ (a set of n labels).

5.3 Public Key Encryption

Indistinguishability under Chosen-Ciphertext Attack (IND-CCA) is a standard assumption for PKE. In the present work the left-or-right formulation by [An01] is used as a base assumption. Figure 5.6a defines the LR-IND-CCA game following Definition 2.2 of [An01].[1] Security according to the left-or-right game is equivalent to security according to the real-or-ideal game defined in Figure 5.6b, as proven by Lemma B.0.2.

$\text{LR-IND-CCA}_\zeta^b$	$\text{RI-IND-CCA}_\zeta^b$
$(\mathsf{sk},\mathsf{pk}) \leftarrow\!\$\ \zeta.\text{gen}()$ $b' \leftarrow \mathcal{A}^{\zeta.\text{enc}(\mathsf{pk},\mathcal{LR}(\cdot,\cdot,b)),\zeta.\text{dec}(\mathsf{sk},\cdot)}(\mathsf{pk})$ **return** b'	$(\mathsf{sk},\mathsf{pk}) \leftarrow\!\$\ \zeta.\text{gen}()$ $b' \leftarrow \mathcal{A}^{\zeta.\text{enc}(\mathsf{pk},\mathcal{LR}(\cdot,0^\lambda,b)),\zeta.\text{dec}(\mathsf{sk},\cdot)}(\mathsf{pk})$ **return** b'
(a)	(b)

Figure 5.6: The LR-IND-CCA and RI-IND-CCA game definitions. The $\mathcal{LR}(\alpha,\beta,0)$ function used in the encryption oracle is defined as α if $b = 0$ and β if $b = 1$. It is mandated that $\mathcal{A}$ never queries the decryption oracle on a ciphertext produced by ζ.enc.

[1] The *global information*, defining constants for the encryption scheme, is assumed to be inherent knowledge in the game.

As illustrated by Figure 5.7a, to transform the traditional game-based definition into a package-based definition a few changes are required. First, $\mathcal{A}$ needs to generate a key pair to work with. To do this, the $\mathsf{PKEY}^{*,1,\zeta}$ package is defined to be an ideal $\mathsf{PKEY}^{1,\zeta}$ package which can only store a single key pair. Second, the oracle calls to ζ.enc and ζ.dec of Figure 5.6 become queries to the ENC and DEC oracles, respectively, provided by the PKE package. Lastly, since PKE expects a handle as an argument, $\mathcal{A}$ has to provide the handle of the key pair that was generated by $\mathsf{PKEY}^{*,1,\zeta}$.

As providing a reduction for this transformation is conceptually straightforward, a formal proof is omitted.

(a) (b)

Figure 5.7: The translation of a traditional game-based definition of IND-CCA (Figure 5.6) to a package-based definition. Note that GETX $:=$ (GET_PK, GET_SK).

Following Lemma G.1 of [BDLF+20], which states that n parallel instances of a game yield a security loss of at most n, the definition of Figure 5.7a can be expanded to a security game where $\mathcal{A}$ can generate and use n key pairs in parallel.

Additionally, the definition of Figure 5.7a needs to be expanded to allow for dishonest key pairs. Dishonest key pairs can be set by $\mathcal{A}$ using a CSET query. From the definition of PKEY.CSET (Figure 2.4) it is clear that $\mathcal{A}$ does not gain any information about honest keys, thus not affecting its distinguishing advantage. Figure 5.7b depicts this parallel composition with dishonest keys.

From Figure 5.7b, a conceptually straightforward transformation from n parallel instances of $\mathsf{PKEY}^{*,1,\zeta}$, which store a single key, to $\mathsf{PKEY}^{1,\zeta}$, which stores many keys, can be made.

Let ζ be a PKE scheme with the syntax defined in Section 2.3, $\mathcal{A}$ an adversary, and GPKE^b_ζ the indistinguishability game defined in Figure 5.8. Then, the PKE advantage is defined as

$$\varepsilon^{\zeta}_{\text{PKE}}(\mathcal{A}) := |\Pr[1 \leftarrow\!\$\ \mathcal{A} \circ \text{GPKE}^0_\zeta] - \Pr[1 \leftarrow\!\$\ \mathcal{A} \circ \text{GPKE}^1_\zeta]|.$$

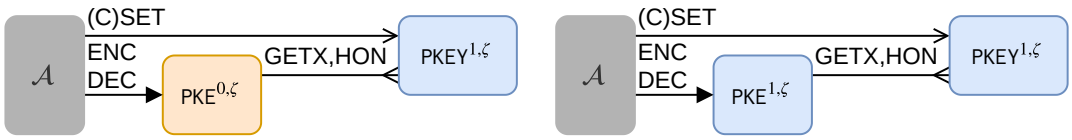


Figure 5.8: The GPKE^{b}_{ζ} game definition. The PKE package is defined in Figure 5.9.

Package Parameters	Package State
b : idealisation bit ζ : PKE primitive	M : **table**$[c \mapsto m]$

(a) The parameters and shared state of a PKE package.

$\mathsf{PKE}^{0,\zeta}.\text{ENC}(h_{\mathsf{pk}}, m)$	$\mathsf{PKE}^{0,\zeta}.\text{DEC}(h_{\mathsf{pk}}, c)$
$\mathsf{pk} \leftarrow \mathsf{PKEY}.\text{GET\_PK}(h_{\mathsf{pk}})$ $c \leftarrow\!\!\$\ \zeta.\text{enc}(\mathsf{pk}, m)$ **return** c	$\mathsf{sk} \leftarrow \mathsf{PKEY}.\text{GET\_SK}(h_{\mathsf{pk}})$ $m \leftarrow \zeta.\text{dec}(\mathsf{sk}, c)$ **if** $[m = \perp]$ **return** $\perp$ **return** m

(b) The real PKE package query implementations.

$\mathsf{PKE}^{1,\zeta}.\text{ENC}(h_{\mathsf{pk}}, m)$	$\mathsf{PKE}^{1,\zeta}.\text{DEC}(h_{\mathsf{pk}}, c)$

$\mathsf{pk} \leftarrow \mathsf{PKEY}.\text{GET\_PK}(h_{\mathsf{pk}})$
if $\mathsf{PKEY}.\text{HON}(h_{\mathsf{pk}})$ **then**
  $c \leftarrow\!\!\$\ \zeta.\text{enc}(\mathsf{pk}, 0^{\lvert m\rvert})$
  $\text{M}[c] \leftarrow m$
else
  $c \leftarrow\!\!\$\ \zeta.\text{enc}(\mathsf{pk}, m)$
end if
return c
	$\mathsf{sk} \leftarrow \mathsf{PKEY}.\text{GET\_SK}(h_{\mathsf{pk}})$
	$m \leftarrow \zeta.\text{dec}(\mathsf{sk}, c)$
	if $[m = \perp]$ **return** $\perp$
	
	if $\mathsf{PKEY}.\text{HON}(h_{\mathsf{pk}})$ **then**
	  $m \leftarrow \text{M}[c]$
	end if
	return m

(c) The ideal PKE package query implementations.

Figure 5.9: Definition of a $\mathsf{PKE}^{b,\zeta}$ package.

5.4 Hybrid Public Key Encryption

Hybrid Public Key Encryption (HPKE) is a special instance of PKE where the plaintext is always a symmetric key [Lip20]. HPKE is the standard Key Encapsulation Method (KEM) construction for PKE if used with a randomly sampled key. This section defines the HPKE assumption that will be used in the present work.

Let ζ be a PKE scheme with the syntax defined in Section 2.3, $\mathcal{A}$ an adversary, and $\text{GHPKE}_{\zeta}^{b,\lambda}$ the indistinguishability game defined in Figure 5.10. Then, the HPKE advantage is defined as

$$\varepsilon_{\text{HPKE}}^{\zeta}(\mathcal{A}) := |\Pr[1 \leftarrow\!\$\ \mathcal{A} \circ \text{GHPKE}_{\zeta}^{0,\lambda}] - \Pr[1 \leftarrow\!\$\ \mathcal{A} \circ \text{GHPKE}_{\zeta}^{1,\lambda}]|,$$

where λ is the size of the keys being encrypted.

By Lemma B.0.3, this advantage can be reduced to the IND-CCA assumption on PKE (Section 5.3).

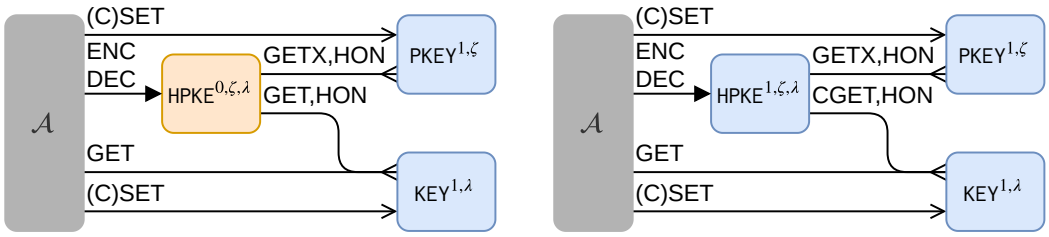


Figure 5.10: The $\text{GHPKE}_{\zeta}^{b,\lambda}$ game definition. The HPKE package is defined in Figure 5.11.

Package Parameters	Package State
b : idealisation bit ζ : PKE primitive λ : secret key size	R : **table**$[\mathsf{k} \mapsto h]$ M : **table**$[c \mapsto h]$

(a) The parameters and shared state of a HPKE package.

$\mathsf{HPKE}^{0,\zeta,\lambda}.\mathrm{ENC}(h_{\mathsf{pk}}, h_{\mathsf{k}})$	$\mathsf{HPKE}^{0,\zeta,\lambda}.\mathrm{DEC}(h_{\mathsf{pk}}, c)$

assert $\mathsf{PKEY}.\mathrm{HON}(h_{\mathsf{pk}}) \vee \neg\mathsf{KEY}.\mathrm{HON}(h_{\mathsf{k}})$
$\mathsf{k} \leftarrow \mathsf{KEY}.\mathrm{GET}(h_{\mathsf{k}})$
$\mathsf{pk} \leftarrow \mathsf{PKEY}.\mathrm{GET\_PK}(h_{\mathsf{pk}})$
$c \leftarrow\!\!\$\ \zeta.\mathrm{enc}(\mathsf{pk}, \mathsf{k})$
$\mathrm{R}[\mathsf{k}] \leftarrow h_{\mathsf{k}}$
return c
	$\mathsf{sk} \leftarrow \mathsf{PKEY}.\mathrm{GET\_SK}(h_{\mathsf{pk}})$
	$m \leftarrow \zeta.\mathrm{dec}(\mathsf{sk}, c)$
	if $[m = \perp]$ **then**
	  return $\perp$
	end if
	return $\mathrm{R}[m]$

(b) The real HPKE package query implementations.

$\mathsf{HPKE}^{1,\zeta,\lambda}.\mathrm{ENC}(h_{\mathsf{pk}}, h_{\mathsf{k}})$	$\mathsf{HPKE}^{1,\zeta,\lambda}.\mathrm{DEC}(h_{\mathsf{pk}}, c)$

assert $\mathsf{PKEY}.\mathrm{HON}(h_{\mathsf{pk}}) \vee \neg\mathsf{KEY}.\mathrm{HON}(h_{\mathsf{k}})$
$\mathsf{pk} \leftarrow \mathsf{PKEY}.\mathrm{GET\_PK}(h_{\mathsf{pk}})$
if $\mathsf{PKEY}.\mathrm{HON}(h_{\mathsf{pk}})$ **then**
  $\mathsf{k} \leftarrow h_{\mathsf{k}}$
  $c \leftarrow\!\!\$\ \zeta.\mathrm{enc}(\mathsf{pk}, 0^{\lambda})$
  $\mathrm{M}[c] \leftarrow h_{\mathsf{k}}$
else
  $\mathsf{k} \leftarrow \mathsf{KEY}.\mathrm{CGET}(h_{\mathsf{k}})$
  $c \leftarrow\!\!\$\ \zeta.\mathrm{enc}(\mathsf{pk}, \mathsf{k})$
end if
$\mathrm{R}[\mathsf{k}] \leftarrow h_{\mathsf{k}}$
return c
	$\mathsf{sk} \leftarrow \mathsf{PKEY}.\mathrm{GET\_SK}(h_{\mathsf{pk}})$
	$m \leftarrow \zeta.\mathrm{dec}(\mathsf{sk}, c)$
	if $[m = \perp]$ **then**
	  return $\perp$
	end if
	if $\mathsf{PKEY}.\mathrm{HON}(h_{\mathsf{pk}})$ **then**
	  $m \leftarrow \mathrm{M}[c]$
	end if
	return $\mathrm{R}[m]$

(c) The ideal HPKE package query implementations.

Figure 5.11: Definition of an $\mathsf{HPKE}^{b,\zeta,\lambda}$ package. Note that the **`assert`** statement in both ENC queries prevents $\mathcal{A}$ from encrypting honest keys with a dishonest key pair, which would allow $\mathcal{A}$ to learn the value of a honest key.

5.4.1 HPKE with DKP

Instead of running a key generation algorithm of a PKE with *implicit* randomness, MLS runs the key generation algorithm with *explicit* randomness. The standard calls this procedure a *Derive Key Pair* (DKP) function. In MLS, the DKP function, defined as a package in Figure 5.13, is run based on pseudorandom coins generated by a KDF. In turn, when the random coins of a DKP are drawn uniformly at random, the behaviour of a DKP is identical to the behaviour of the key generation algorithm with implicit randomness, as proven in Lemma B.0.4.

Let ζ be a PKE scheme with the syntax defined in Section 2.3, $\mathcal{A}$ an adversary, and $\text{GHPKE2}_{\zeta}^{b,\lambda,\lambda_S}$ the indistinguishability game defined in Figure 5.12. Then, the HPKE with DKP advantage is defined as

$$\varepsilon^{\zeta}_{\text{HPKE2}}(\mathcal{A}) := |\Pr[1 \leftarrow\!\$\ \mathcal{A} \circ \text{GHPKE2}_{\zeta}^{0,\lambda,\lambda_S}] - \Pr[1 \leftarrow\!\$\ \mathcal{A} \circ \text{GHPKE2}_{\zeta}^{1,\lambda,\lambda_S}]|,$$

where λ the size of the keys being encrypted, λ_S the size of key pair seeds. Then, It holds that

$$\varepsilon^{\zeta}_{\text{HPKE2}}(\mathcal{A}) = \varepsilon^{\zeta}_{\text{HPKE}}(\mathcal{A} \circ \mathcal{R}),$$

for $\mathcal{R}$ emulating the interface of the DKP package using the implementation of $\text{PKEY}^{1,\zeta}$.

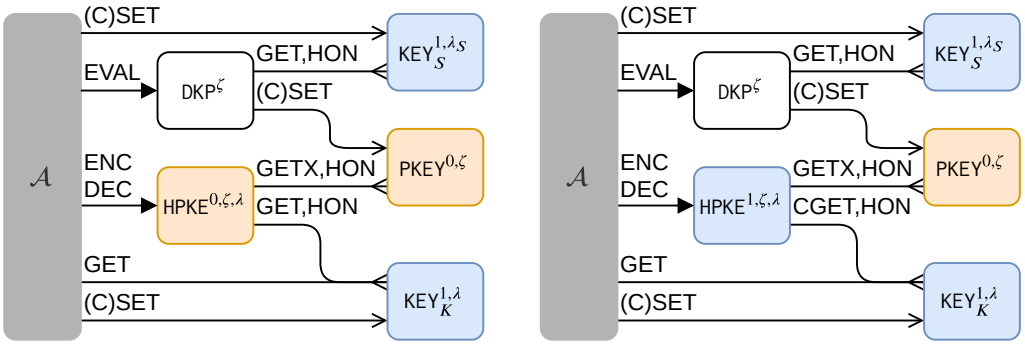


Figure 5.12: The GHPKE2 game definition. The DKP package is defined in Figure 5.13.

Package Parameters	Package State
ζ : PKE primitive	-

(a) The parameters and shared state of a DKP package.

$\mathsf{DKP}^{\zeta}.\text{EVAL}(h_s)$

$s \leftarrow \mathsf{KEY}_S.\text{GET}(h_s)$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \zeta.\text{gen}(s)$
if $\mathsf{KEY}_S.\text{HON}(h_s)$ **then**
    return $\mathsf{PKEY}.\text{SET}(\mathsf{sk}, \mathsf{pk})$
else
    return $\mathsf{PKEY}.\text{CSET}(\mathsf{sk}, \mathsf{pk})$
end if

(b) The DKP package query implementation.

Figure 5.13: Definition of a DKP^{ζ} package.

Chapter 6

TreeKEM Security

This chapter defines and proves the security of MLS's TreeKEM component. Recall from Section 3.1 that TreeKEM aims to provide a shared secret for all Members of a Group. Every Member knows a secret value of one of the leaves of the TreeKEM tree. Using the secret values along their direct path, each Member can derive the same secret at the tree root as any other Group Member. Therefore, the security goal defined in Section 6.1 requires that the value at the root of a TreeKEM tree is indistinguishable from a random value if none of the leaf secrets is known to the adversary.

Section 6.1 defines the security goal for TreeKEM. Next, Section 6.2 introduces the packages used to model and compose a TreeKEM tree. Lastly, Section 6.3 and Section 6.4 provide a security proof for TreeKEM based on the assumption introduced in Chapter 5.

The State Separating Proofs model of TreeKEM presented in this chapter is based on the implementations of Π.Commit (Algorithm 1), Π.ReceiveCommit (Algorithm 2), and Π^*.CreateDirectPath (Algorithm 7).

Preliminaries Let n be the number of nodes in the tree and m the number of layers in (i.e. height of) the tree. For consistency, i is used for node indices and j is used for layer indices. $p(i)$ is the index of a node's parent node and $\mathcal{L}_j$ is the set of indices of nodes at layer j. Layer $j = 0$ is the layer of the tree root.

6.1 Security Goal

The security goal of TreeKEM can be formulated as an indistinguishability game, $\text{GTREE}^{b,\lambda,m}_{\kappa,\zeta}$, defined in Figure 6.1. Then, for all $\mathcal{A}$ the TreeKEM advantage is defined as

$$\varepsilon^{\kappa,\zeta}_{\text{TreeKEM}}(\mathcal{A}) := |\Pr[1 \leftarrow\!\$\ \mathcal{A} \circ \text{GTREE}^{0,\lambda,m}_{\kappa,\zeta}] - \Pr[1 \leftarrow\!\$\ \mathcal{A} \circ \text{GTREE}^{1,\lambda,m}_{\kappa,\zeta}]|,$$

where κ is a KDF, ζ is a PKE scheme, and λ is the length of keys consumed and outputted by κ and encrypted by HPKE. More specifically, this chapter will proof Theorem 6.4.1.

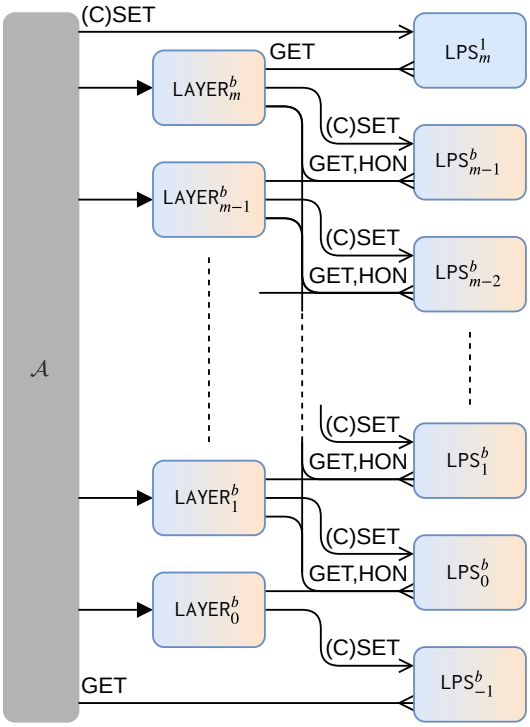
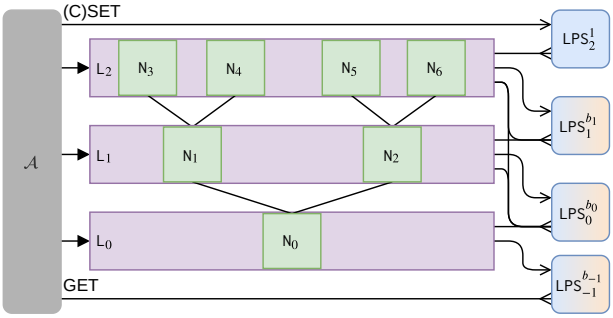


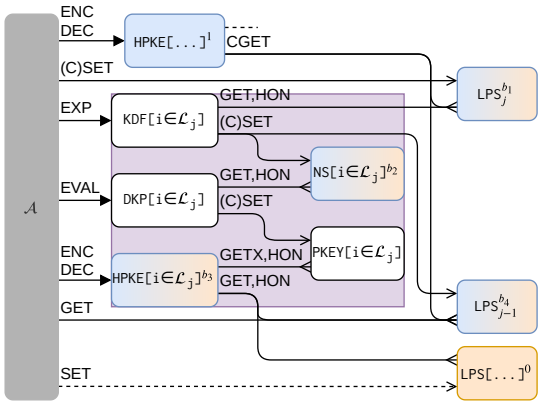
Figure 6.1: The $\text{GTREE}^{b,\lambda,m}_{\kappa,\zeta}$ game definition. The packages are defined in Section 6.2. The leaf layer is at the top of this graph. The oracle calls from $\mathcal{A}$ to the LAYER packages represent oracle calls to all nodes, as illustrated by Figure 6.2.

6.2 Package Definitions

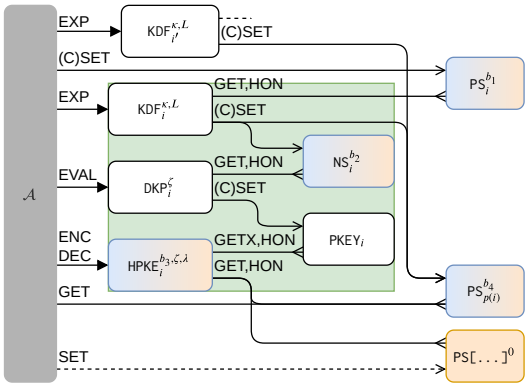
The TreeKEM tree is modelled as a binary tree consisting of m layers and n nodes. This section introduces the packages that the tree consists of. Figure 6.2 gives an overview of the packages and their composition.



(a) An example TREE package for a 4-member Group. In this figure $\mathsf{N}_i = \mathsf{NODE}_i$ and $\mathsf{L}_j = \mathsf{LAYER}_j$.



(b) The composition of a $\mathsf{LAYER}_j^{b_1 b_2 b_3 b_4}$ package and its relations to other layers.



(c) The composition of a $\mathsf{NODE}_i^{b_1 b_2 b_3 b_4}$ package and its relations to other nodes.

Figure 6.2: Overview of the TreeKEM package modelling. Coloured non-rounded squares are used to indicate how the subfigures relate to each other. Dashed lines are used to indicate the existence of additional packages that fall outside the scope of the figure.

Path Secret Packages Let a per-node path secret package, PS, seen in Figure 6.2c, be defined as

$$\mathsf{PS}_i^b := \mathsf{KEY}^{b,\lambda},$$

and let a per-layer path secret package, LPS, seen in Figure 6.2b, be defined as the parallel composition of all PS packages for that layer

$$\forall j \in \{0, \ldots, m\} : \mathsf{LPS}_j^b := \mathsf{PS}[i \in \mathcal{L}_j]^b.$$

A PS package represents the path secret (see Section 3.1) for a single node. It stores the path secret so it can be accessed by other packages while remaining secret to $\mathcal{A}$. On the

other hand, the LPS package is a purely abstract package that serves to ease the proof by modelling all path secrets of one layer. This modelling choice is motivated by the fact that all per-node path secrets in one layer are independent and can, therefore, be considered in parallel.

Node Package Let a NODE package be defined as per Figure 6.2c. Nodes are indexed by their position in the tree as NODE_i. For every node NODE_i let the node secret package, NS_i^b, be defined as

$$\mathsf{NS}_i^b := \mathsf{KEY}^{b,\lambda},$$

let the node's public-private key pair package, PKEY_i, be defined as

$$\mathsf{PKEY}_i := \mathsf{PKEY}^{0,\zeta},$$

and let the KDF be defined as a KDF package (Figure 5.4) for $n = 2$ with $L := \{\texttt{"path"}, \texttt{"node"}\}$. The idealisation bits of a $\mathsf{NODE}_i^{b_1 b_2 b_3 b_4}$ package have the following correspondence: $\mathsf{PS}_i^{b_1}$, $\mathsf{NS}_i^{b_2}$, $\mathsf{HPKE}_i^{b_3}$, and $\mathsf{PS}_i^{b_4}$. When composing nodes, it must hold that $b_4 = b_1$ for $\mathsf{NODE}_i^{xxxb_4}$ and $\mathsf{NODE}_{p(i)}^{b_1 xxx}$ for the composition to be well-defined.

The NODE package represents the computations that are executed from the perspective of a single node in the tree (i.e. leaf node or intermediary node). This roughly corresponds to the behaviour encoded in Algorithm 1.

Layer Package Let a LAYER package be defined as per Figure 6.2b. Layers are indexed by their depth j in the tree as LAYER_j. Intuitively, LAYER_j is the parallel composition of the NODE packages in layer j, i.e. $\mathsf{NODE}[i \in \mathcal{L}_j]$. The idealisation bits in $\mathsf{LAYER}_j^{b_1 b_2 b_3 b_4}$ correspond to the idealisation bits of $\mathsf{LPS}_j^{b_1}$, $\mathsf{NS}[i \in \mathcal{L}_j]^{b_2}$, $\mathsf{HPKE}[i \in \mathcal{L}_j]^{b_3}$, and $\mathsf{LPS}_{j-1}^{b_4}$ respectively. When composing layers it must hold that $b_4 = b_1$ for layers $\mathsf{LAYER}_j^{xxxb_4}$ and $\mathsf{LAYER}_{j-1}^{b_1 xxx}$ for the composition to be well-defined.

A layer j is called *real* if its bits are $\mathsf{LAYER}_j^{b_1 000}$ for $b_1 \in \{0, 1\}$. A layer j is called *ideal* if its bits are LAYER_j^{1111}

Similar to the LPS the LAYER package is an abstract representation of all nodes on the same layer in the TreeKEM tree and is used to ease the proof.

6.3 Layer Idealisation

The first part of the security proof of TreeKEM is to show that for all j, LAYER_j can be idealised if LPS_j is ideal as a starting point, i.e. $b_1 = 1$. Given this idealisation proof, the security of an entire TreeKEM tree can be proven using a hybrid argument over the layers.

Theorem 6.3.1. (Layer Security) Let $\mathcal{A}$ be an adversary, then $\forall j \in \{0, \ldots, m\}$, it holds that

$$\varepsilon(\mathcal{A}; \mathsf{LAYER}_j^{1000}, \mathsf{LAYER}_j^{1111}) \leq 2^j(\varepsilon^{\kappa}_{\mathrm{KDF}}(\mathcal{A} \circ \mathcal{R}_{0,j}) + \varepsilon^{\zeta}_{\mathrm{HPKE}}(\mathcal{A} \circ \mathcal{R}_{1,j})),$$

where $\mathcal{R}_{0,j}$ and $\mathcal{R}_{1,j}$ are defined in Lemma 6.3.2 and Lemma 6.3.3 respectively.

Proof The proof of Theorem 6.3.1 consists of two steps. In the first step, part of the layer is idealised based on the GKDF game. In the second step, the layer idealisation is finished based on the GHPKE game. Figure 6.3 presents a graphical overview of the proof.

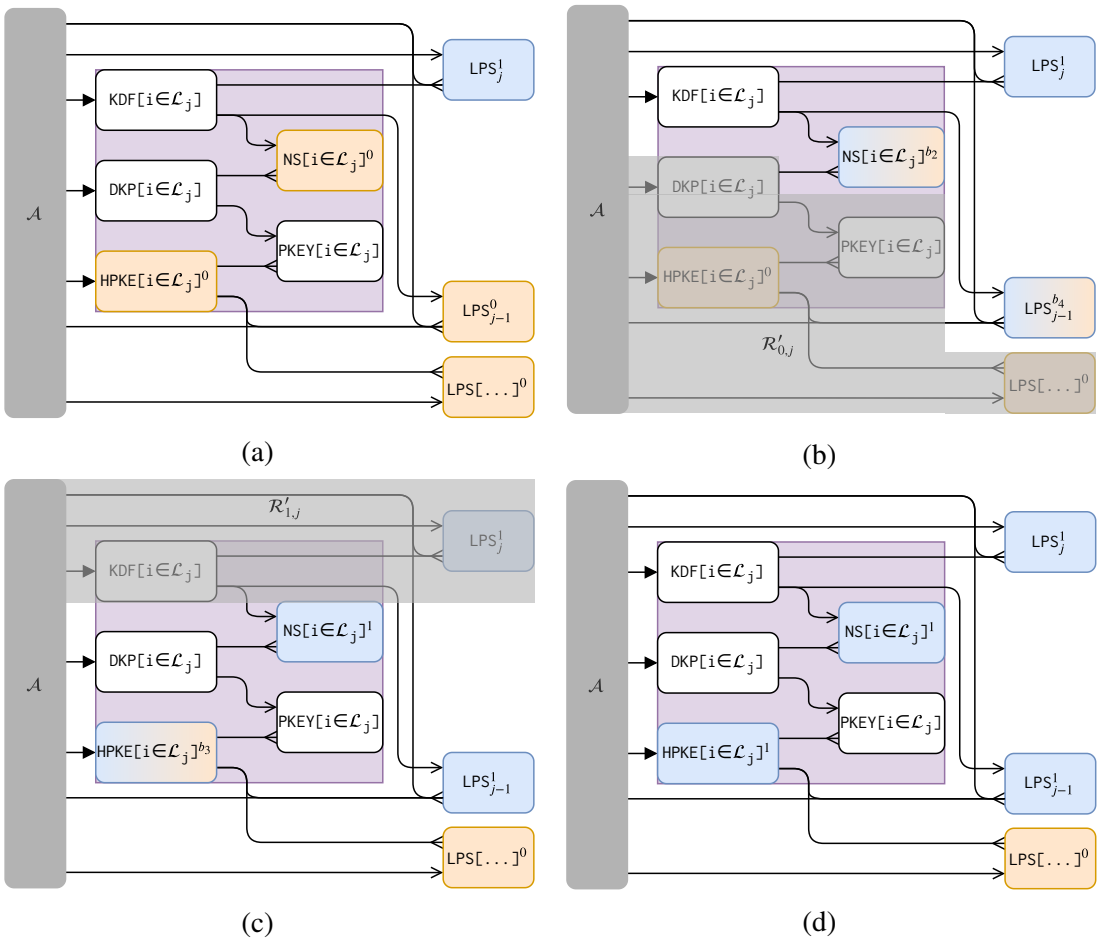


Figure 6.3: Overview of the LAYER packages idealisation proof. **(a)** shows a real LAYER_j^{1000} package based on an ideal LPS. **(b)** shows an indistinguishability game for b_2 and b_4. **(c)** shows an indistinguishability game for b_3. **(d)** shows an idealised LAYER_j^{1111} package. $\mathsf{LPS}[\ldots]^0$ represents the LPS packages of all ancestor layers of layer $j-1$.

Lemma 6.3.2. $\forall \mathcal{A}$ and $\forall j \in \{0, \ldots, m\}$, it holds that

$$\varepsilon(\mathcal{A}; \mathsf{LAYER}_j^{1000}, \mathsf{LAYER}_j^{1101}) \leq 2^j \cdot \varepsilon_{\text{KDF}}^{\kappa}(\mathcal{A} \circ (\mathcal{T}_{0,j} \circ \mathcal{R}'_{0,j})),$$

where $\mathcal{R}'_{0,j}$ is defined as the grey area in Figure 6.3b and $\mathcal{T}_{0,j}$ is defined as sampling $i \leftarrow\!\!\$\ \{0, \ldots, 2^j\}$ uniformly at random and exposing only the corresponding $(\mathsf{KDF}_i, \mathsf{NS}_i)$ packages. Then, let $\mathcal{R}_{0,j} := (\mathcal{T}_{0,j} \circ \mathcal{R}'_{0,j})$.

Proof The proof of Lemma 6.3.2 is based on the GKDF game (Section 5.2, for $n = 2$). Note that both b_2 and b_4 correspond to the single idealisation bit of GKDF. Figure 6.3b shows $(\mathcal{A} \circ \mathcal{R}'_{0,j})$, which interacts with 2^j parallel instances of GKDF. Then, $\mathcal{T}_{0,j}$ is defined to restrict the adversary game to a single GKDF instance. Given 2^j possible behaviours for $\mathcal{T}_{0,j}$, the advantage of $\mathcal{A}$ in $\mathsf{LAYER}_j^{1b_20b_4}$ is bounded by 2^j times the reduction adversary's (composed of $\mathcal{A}$, $\mathcal{R}_{0,j}$, and $\mathcal{T}_{0,j}$) advantage in $\text{GKDF}_{\kappa}^{b,2,\lambda_s,\lambda_t}$.

□

Lemma 6.3.3. $\forall \mathcal{A}$ and $\forall j \in \{0, \ldots, m\}$, it holds that

$$\varepsilon(\mathcal{A}; \mathsf{LAYER}_j^{1101}, \mathsf{LAYER}_j^{1111}) \leq 2^j \cdot \varepsilon_{\text{HPKE}}^{\zeta}(\mathcal{A} \circ (\mathcal{T}_{1,j} \circ \mathcal{R}'_{1,j})),$$

where $\mathcal{R}'_{1,j}$ is defined as the grey area in Figure 6.3c and $\mathcal{T}_{1,j}$ is defined as sampling $i \leftarrow\!\!\$\ \{0, \ldots, 2^j\}$ uniformly at random and exposing only the corresponding $(\mathsf{NS}_i, \mathsf{DKP}_i, \mathsf{PKEY}_i, \mathsf{HPKE}_i)$ packages. Then, let $\mathcal{R}_{1,j} := (\mathcal{T}_{1,j} \circ \mathcal{R}'_{1,j})$.

Proof The proof of Lemma 6.3.3 is based on the GHPKE game (Section 5.4). Figure 6.3c shows $(\mathcal{A} \circ \mathcal{R}'_{1,j})$, which interacts with 2^j parallel instances of GHPKE. Then, $\mathcal{T}_{1,j}$ is defined to restrict the adversary game to a single GHPKE instance. Given 2^j possible behaviours for $\mathcal{T}_{1,j}$, the advantage of $\mathcal{A}$ in $\mathsf{LAYER}_j^{11b_31}$ is bounded by 2^j times the reduction adversary's (composed of $\mathcal{A}$, $\mathcal{R}_{1,j}$, and $\mathcal{T}_{1,j}$) advantage in $\text{GHPKE}_{\zeta}^{b,\lambda}$.

□

By the combination of the results of Lemma 6.3.2 and Lemma 6.3.3 an upper bound $\forall \mathcal{A}$ and $\forall j \in \{0, \ldots, m\}$ is obtained, namely:

$$\begin{aligned}\varepsilon(\mathcal{A}; \mathsf{LAYER}_j^{1000}, \mathsf{LAYER}_j^{1111}) &\leq 2^j \cdot \varepsilon_{\text{KDF}}^{\kappa}(\mathcal{A} \circ \mathcal{R}_{0,j}) + 2^j \cdot \varepsilon_{\text{HPKE}}^{\zeta}(\mathcal{A} \circ \mathcal{R}_{1,j}) \\ &= 2^j(\varepsilon_{\text{KDF}}^{\kappa}(\mathcal{A} \circ \mathcal{R}_{0,j}) + \varepsilon_{\text{HPKE}}^{\zeta}(\mathcal{A} \circ \mathcal{R}_{1,j})),\end{aligned}$$

proving Theorem 6.3.1.

□

6.4 Tree Idealisation

The second part of the security proof of TreeKEM uses Theorem 6.3.1 and a hybrid argument to idealise every layer in a TreeKEM tree step-by-step. Figure 6.4 visualises the proof in this section.

Theorem 6.4.1. (TreeKEM Security) Let $\mathcal{A}$ be an adversary and $\text{GTREE}^{b,\lambda,m}_{\kappa,\zeta} = \mathsf{TREE}^{b\ldots b}$. Then it holds that

$$\begin{aligned}\varepsilon^{\kappa,\zeta}_{\text{TreeKEM}}(\mathcal{A}) &= \varepsilon(\mathcal{A}; \mathsf{TREE}^{0\ldots0}, \mathsf{TREE}^{1\ldots1}) \\ &\leq \sum_{j=0}^{m} \varepsilon(\mathcal{A} \circ \mathcal{R}_{m-j}; \mathsf{LAYER}^{1000}_{j}, \mathsf{LAYER}^{1111}_{j}),\end{aligned}$$

where $\mathcal{R}_x$s are defined as illustrated by the various grey areas in Figure 6.4.

Proof To start, as shown in Figure 6.4b, the leaf layer of the tree is idealised. This relies on the fact that the path secrets for leaf nodes, LPS_0, are ideal by default. These can be considered ideal because they are never exposed to $\mathcal{A}$ and only used to derive other values in the tree. Hence, using Theorem 6.3.1 for $j = m$, the leaf layer is idealised, giving:

$$\varepsilon(\mathcal{A}; \mathsf{TREE}^{00\ldots0}, \mathsf{TREE}^{10\ldots0}) \leq \varepsilon(\mathcal{A} \circ \mathcal{R}_0; \mathsf{LAYER}^{1000}_{m}, \mathsf{LAYER}^{1111}_{m})$$

where $\mathcal{R}_0$ is defined as the grey area in Figure 6.4b.

With the path secrets of layer $j = (m-1)$ ideal as a result of idealising layer $j = m$, layer $j = (m-1)$ can be idealised using Theorem 6.3.1, giving:

$$\varepsilon(\mathcal{A}; \mathsf{TREE}^{100\ldots0}, \mathsf{TREE}^{110\ldots0}) \leq \varepsilon(\mathcal{A} \circ \mathcal{R}_1; \mathsf{LAYER}^{1000}_{m-1}, \mathsf{LAYER}^{1111}_{m-1}),$$

where $\mathcal{R}_1$ is defined as the grey area in Figure 6.4c. This process can be repeated for all layers $j \in \{(m-2), \ldots, 2\}$. Lemma 38 from [BDLF+18] allows for the use of a hybrid argument to arrive at the tree state shown in Figure 6.4d. With the path secrets of layer $j = 1$ idealised, LPS^1_1, Theorem 6.3.1 can be used to obtain:

$$\varepsilon(\mathcal{A}; \mathsf{TREE}^{1\ldots100}, \mathsf{TREE}^{1\ldots110}) \leq \varepsilon(\mathcal{A} \circ \mathcal{R}_{m-1}; \mathsf{LAYER}^{1000}_{1}, \mathsf{LAYER}^{1111}_{1}),$$

where $\mathcal{R}_{m-1}$ is defined as the grey area in Figure 6.4e.

Finally, with the path secrets of layer $j = 0$, Theorem 6.3.1 can be applied one last time to idealise the root layer and its path secrets, giving:

$$\varepsilon(\mathcal{A}; \mathsf{TREE}^{1\ldots10}, \mathsf{TREE}^{1\ldots11}) \leq \varepsilon(\mathcal{A} \circ \mathcal{R}_m; \mathsf{LAYER}^{1000}_{0}, \mathsf{LAYER}^{1111}_{0}),$$

where $\mathcal{R}_m$ is defined as the grey area in Figure 6.4f.

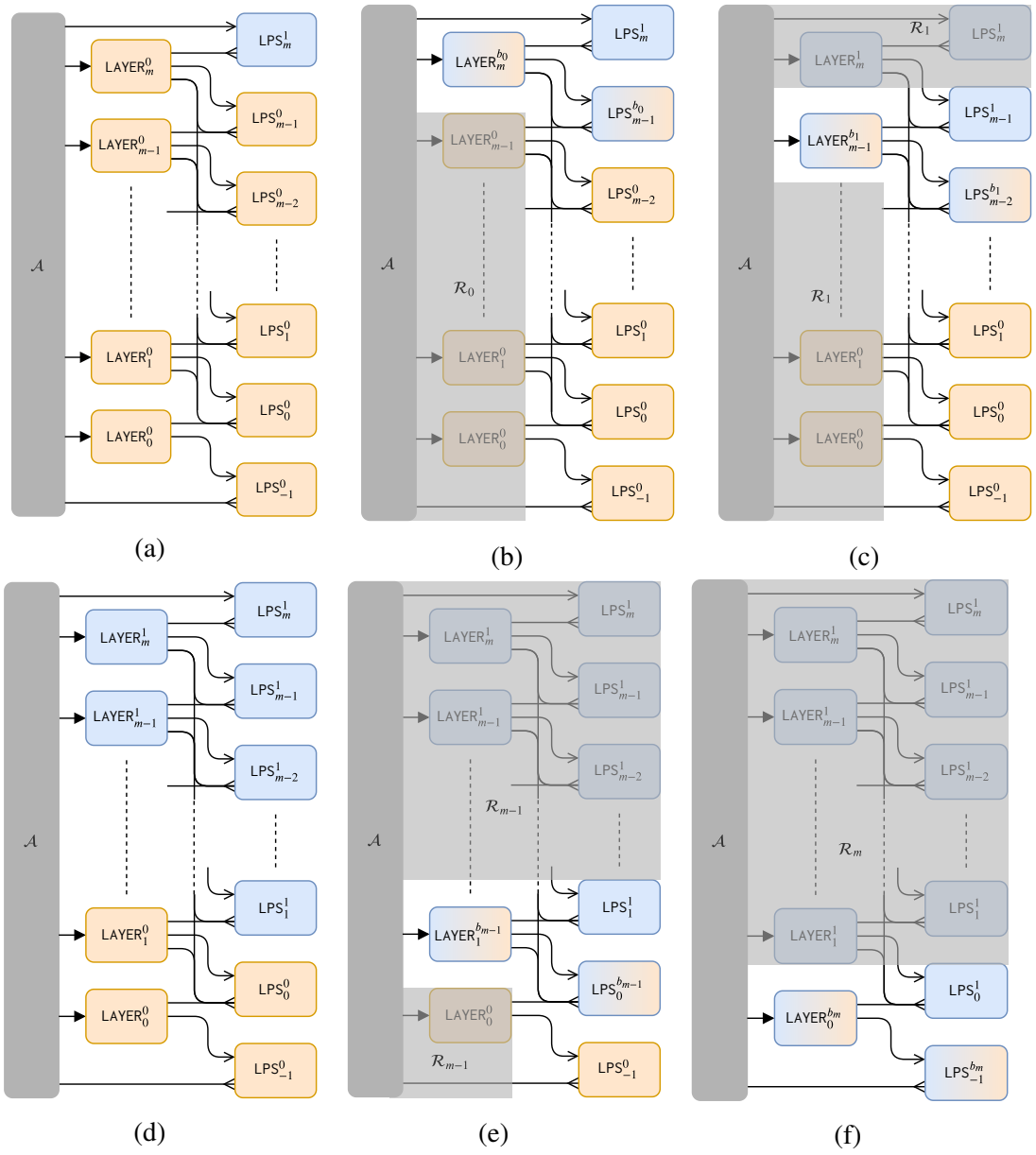


Figure 6.4: Overview of the TREE package idealisation proof.

By composing the results obtained in this section (again, using Lemma 38 from [BDLF$^+$18]), an upper bound $\forall \mathcal{A}$ is obtained, namely:

$$\begin{aligned}
\varepsilon(\mathcal{A}; \mathsf{TREE}^{0\ldots0}, \mathsf{TREE}^{1\ldots1}) &= \varepsilon(\mathcal{A}; \mathsf{TREE}^{00\ldots0}, \mathsf{TREE}^{10\ldots0}) \\
&\quad + \ldots \\
&\quad + \varepsilon(\mathcal{A}; \mathsf{TREE}^{1\ldots10}, \mathsf{TREE}^{1\ldots11}) \\
&\leq \varepsilon(\mathcal{A} \circ \mathcal{R}_0; \mathsf{LAYER}_0^{1000}, \mathsf{LAYER}_0^{1111}) \\
&\quad + \ldots \\
&\quad + \varepsilon(\mathcal{A} \circ \mathcal{R}_m; \mathsf{LAYER}_m^{1000}, \mathsf{LAYER}_m^{1111}) \\
&= \sum_{j=0}^{m} \varepsilon(\mathcal{A} \circ \mathcal{R}_{m-j}; \mathsf{LAYER}_j^{1000}, \mathsf{LAYER}_j^{1111}),
\end{aligned}$$

proving Theorem 6.4.1 and by extension the security goal of this chapter.

□

Chapter 7

Key Schedule Security

This chapter defines and proves the security of the key schedule of MLS. Recall from Section 3.3 that the key schedule exists to derive various keys for different purposes from two secret values shared by a Group. The *init secret* of the previous epoch and the *update secret* (the value at the root of the TreeKEM tree) are put through a PRF and KDF to obtain keys for the next epoch.

First, Section 7.1 defines the security goal of the key schedule. Next, Section 7.2 introduces the packages that compose the key schedule. Lastly, Section 7.3 provides a security proof for the key schedule.

The State Separating Proofs model of the key schedule is based largely on the implementation of Π^*.ComputeGroupKeys (Algorithm 6).

7.1 Security Goal

The security goal of the key schedule can be formulated as an indistinguishability game, $\text{GKS}^{b,\lambda_K,\lambda_S,\lambda,n}_{\kappa,\varphi}$, defined in Figure 7.1. Then, for all $\mathcal{A}$ the key schedule advantage is defined as

$$\varepsilon^{\kappa,\varphi}_{\text{KS}}(\mathcal{A}) := |\Pr[1 \leftarrow\!\$\ \mathcal{A} \circ \text{GKS}^{0,\lambda_K,\lambda_S,\lambda,n}_{\kappa,\varphi}] - \Pr[1 \leftarrow\!\$\ \mathcal{A} \circ \text{GKS}^{1,\lambda_K,\lambda_S,\lambda,n}_{\kappa,\varphi}]|,$$

where κ is a KDF, φ a PRF, λ_K the size of keys outputted by κ, λ_S the size of epoch secrets, λ the size of keys in LPS, and n the number of epochs of a Group.

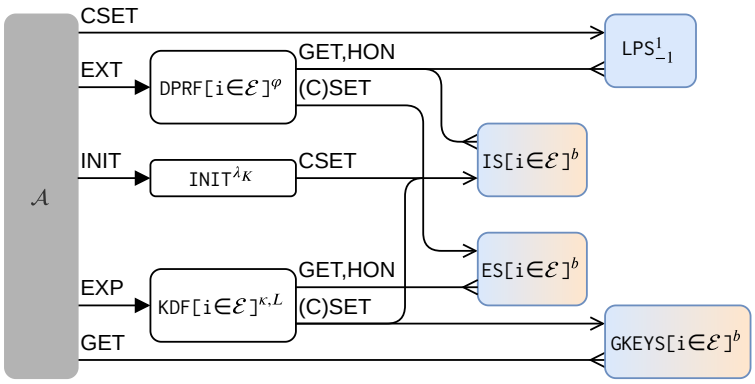


Figure 7.1: The $\text{GKS}^{b,\lambda_K,\lambda_S,\lambda,n}_{\kappa,\varphi}$ game definition, where $\mathcal{E} := \{0,\ldots,n\}$. The packages are defined in Section 7.2, except for the LPS package which is defined in Section 6.2.

7.2 Package Definitions

The key schedule is modelled as the composition of various packages shown in Figure 7.1, which is based primarily on Algorithm 6. The LPS package is defined in Section 6.2, and the DPRF package is defined in Section 5.1. The remaining packages are defined in this section.

Secrets Packages The init secret package, IS, is defined as

$$\text{IS}^b := \text{KEY}^{b,\lambda_K},$$

and, similarly, the epoch secret package, ES, is defined as

$$\text{ES}^b := \text{KEY}^{b,\lambda_S}.$$

Recall from Section 3.3 that the init secret and epoch secret are used, together with the update secret, as a ratcheting mechanism for the key schedule in MLS.

Init Package The init package, INIT^{λ_K}, is a package that must be called to initialise IS^b_0 with the initial *init secret* of the Group as specified by the RFC.

Calling the INIT package once will result in the call $\text{IS}_0.\text{CSET}(\iota)$, where ι is the first *init secret* for a Group as defined by the RFC. This will return the handle h corresponding to that call. Calling INIT again will do nothing and only return the same handle h.

Group Keys Package The package for group keys, GKEYS, is a parallel composition of 4 KEY packages

$$\mathsf{GKEYS}^b := \mathsf{KEY}[i \in G]^{b,\lambda_K},$$

where $G := \{0, 1, 2, 3\}$ represents the sender data secret, handshake secret, application secret, and confirmation key. Recall from Section 3.3 that these are the keys, besides the init secret and epoch secret, defined for the key schedule.

KDF Package From the group keys KEY packages it follows that a total of 5 keys (the init secret, sender data secret, handshake secret, and application secret) are derived from one value (the epoch secret). Therefore, a KDF with five output values is defined as a KDF package for $n = 5$ with

$$L := \{\texttt{"sender data"}, \texttt{"handshake"}, \texttt{"app"}, \texttt{"confirm"}, \texttt{"init"}\}.$$

7.3 Key Schedule Idealisation

Careful inspection of Figure 7.1 reveals a circular idealisation dependency. The ES package can be idealised based on the GDPRF game (Section 5.1). But for that assumption the IS package should be ideal. The IS package can be idealised based on the GKDF game (Section 5.2). However, to use that assumption the ES package must be ideal. Hence, the security of the key schedule cannot be proven directly.

Instead, the key schedule is "unrolled" to prove its security. Figure 7.2a illustrates the unrolled key schedule. The security proof consists of two parts. In the first part, it is shown that an adversary against the unrolled key schedule is an adversary against $\text{GKS}^{b,\lambda_K,\lambda_S,\lambda,n}_{\kappa,\varphi}$. In the second part the security of the unrolled key schedule is proven.

Lemma 7.3.1. (Unrolled Key Schedule) Let $\mathcal{A}$ be an adversary and let n be the number of epochs for a group and let $\mathsf{UKS}^{b_0 \ldots b_n}$ be defined as illustrated in Figure 7.2a. Then, it holds that

$$\varepsilon^{\kappa,\varphi}_{\text{KS}}(\mathcal{A}) = \varepsilon(\mathcal{A}; \mathsf{UKS}^{0\ldots0}, \mathsf{UKS}^{1\ldots1}).$$

Proof Recall the design of the key schedule from Section 3.3. The first epoch secret is derived from the first init secret and the second init secret is derived from the first epoch secret, etc. In the context of $\text{GKS}^{b,\lambda_K,\lambda_S,\lambda,n}_{\kappa,\varphi}$ this means that ES_0 is computed by DPRF_0 from IS_0 and that IS_1 is computed by KDF from ES_0, etc. This, by design, is the pattern of UKS seen in Figure 7.2a. Therefore, the advantage of $\mathcal{A}$ in $\text{GKS}^{b,\lambda_K,\lambda_S,\lambda,n}_{\kappa,\varphi}$ is equivalent to that of $\mathcal{A}$ against $\mathsf{UKS}^{b\ldots b}$.

□

Theorem 7.3.2. (Key Schedule Security) Let $\mathcal{A}$ be an adversary, n be the number of epochs for a group, and $\mathsf{UKS}^{b_0 \dots b_n}$ be defined as illustrated in Figure 7.2a. Then, it holds that

$$\varepsilon(\mathcal{A}; \mathsf{UKS}^{0\dots0}, \mathsf{UKS}^{1\dots1}) \leq \sum_{i=0}^{n} \varepsilon^{\varphi}_{\text{DPRF}}(\mathcal{A} \circ \mathcal{R}_{2i}) + \varepsilon^{\kappa}_{\text{KDF}}(\mathcal{A} \circ \mathcal{R}_{2i+1}),$$

where, for all i, $\mathcal{R}_{2i}$ and $\mathcal{R}_{2i+1}$ are defined as illustrated in Figure 7.2.

Proof The first step of the proof of Theorem 7.3.2 is to idealise the ES_0 package based on the GDPRF game (Section 5.1). This first step is possible because both the IS_0 and LPS package are ideal, the former by definition and the latter by Theorem 6.4.1. Figure 7.2b shows $(\mathcal{A} \circ \mathcal{R}_0)$ interacting with the same packages as the adversary in Figure 5.1. Therefore, it holds that

$$\varepsilon(\mathcal{A}; \mathsf{UKS}^{00\dots0}, \mathsf{UKS}^{10\dots0}) \leq \varepsilon^{\varphi}_{\text{DPRF}}(\mathcal{A} \circ \mathcal{R}_0).$$

Now the ES_0 package is ideal. Thus, the next step is to idealise the IS_1 based on the GKDF game (Section 5.2 for $n = 5$). Figure 7.2c shows $(\mathcal{A} \circ \mathcal{R}_1)$ interacting with the same packages as the adversary in Figure 5.3. Therefore, it holds that

$$\varepsilon(\mathcal{A}; \mathsf{UKS}^{100\dots0}, \mathsf{UKS}^{110\dots0}) \leq \varepsilon^{\kappa}_{\text{KDF}}(\mathcal{A} \circ \mathcal{R}_1).$$

Now the IS_1 package is ideal. Next, analogously to the first step of the proof, the ES_1 is idealised based on the GDPRF game (Section 5.1). Notice that this time the honesty of the value in ES_1 depends on whether the keys stored in the IS_1 package and LPS package was honest. Figure 7.2d shows $(\mathcal{A} \circ \mathcal{R}_2)$ interacting with the same packages as the adversary in Figure 5.1. Therefore, it holds that

$$\varepsilon(\mathcal{A}; \mathsf{UKS}^{1100\dots0}, \mathsf{UKS}^{1110\dots0}) \leq \varepsilon^{\varphi}_{\text{DPRF}}(\mathcal{A} \circ \mathcal{R}_2).$$

The proof proceeds via a hybrid argument over the number of epochs n that the Group exists. Hence, the overall advantage of an adversary in the UKS game is

$$\begin{aligned}
\varepsilon(\mathcal{A}; \mathsf{UKS}^{0\dots0}, \mathsf{UKS}^{1\dots1}) &\leq \varepsilon(\mathcal{A}; \mathsf{UKS}^{00\dots0}, \mathsf{UKS}^{10\dots0}) \\
&\quad + \dots \\
&\quad + \varepsilon(\mathcal{A}; \mathsf{UKS}^{1\dots10}, \mathsf{UKS}^{1\dots11}) \\
&\leq \varepsilon^{\varphi}_{\text{DPRF}}(\mathcal{A} \circ \mathcal{R}_0) + \varepsilon^{\kappa}_{\text{KDF}}(\mathcal{A} \circ \mathcal{R}_1) \\
&\quad + \dots \\
&\quad + \varepsilon^{\varphi}_{\text{DPRF}}(\mathcal{A} \circ \mathcal{R}_{2n}) + \varepsilon^{\kappa}_{\text{KDF}}(\mathcal{A} \circ \mathcal{R}_{2n+1}) \\
&= \sum_{i=0}^{n} \varepsilon^{\varphi}_{\text{DPRF}}(\mathcal{A} \circ \mathcal{R}_{2i}) + \varepsilon^{\kappa}_{\text{KDF}}(\mathcal{A} \circ \mathcal{R}_{2i+1}),
\end{aligned}$$

proving Theorem 7.3.2 and by extension the security goal of this section.

□

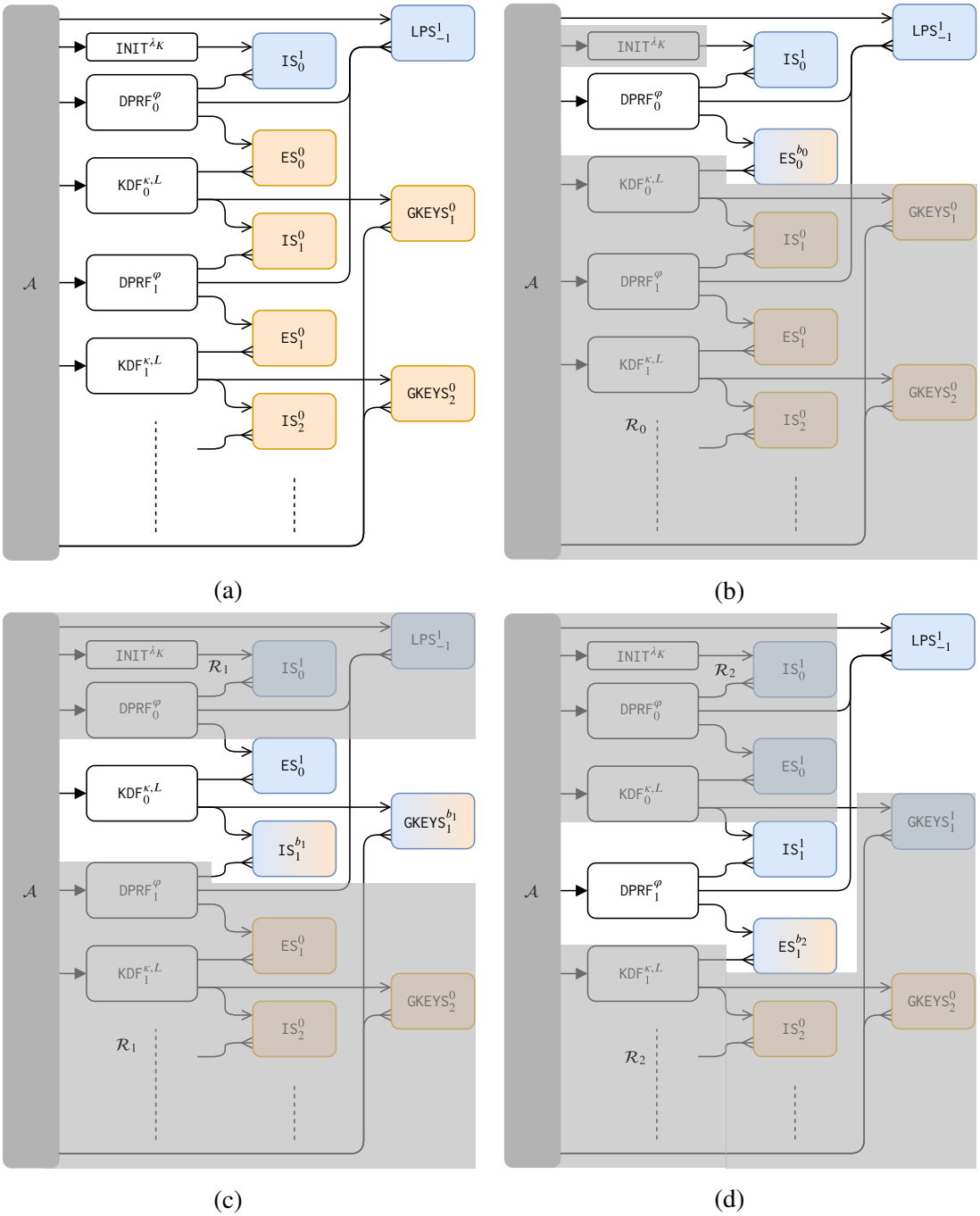


Figure 7.2: Overview of the UKS package idealisation proof.

Chapter 8

Discussion

The present work gives the first State Separating Proofs based cryptographic analysis of the Message Layer Security (MLS) protocol RFC. The analysis covers TreeKEM and the key schedule as found in draft 8 of the RFC and shows both are secure in the static adversarial model using reasonable assumptions on Pseudorandom Functions, Key Derivation Functions, and Public-Key Encryption schemes.

Here, "secure" means that an adversary without knowledge of any of the key shares has a bounded distinguishing advantage for honestly generated keys and randomly sampled keys that are used to secure communication in MLS. In other words, TreeKEM and the key schedule produce "good" keys. This fact may be used to prove the security of the MLS protocol at large. Also, the SSP model that was created may be used to show that the key schedule provides FS and PCS.

This chapter provides a discussion on the limitations of and contributions to the methodology used in this thesis (Section 8.1) and a collection of possible future directions of research (Section 8.2).

8.1 Methodology

8.1.1 Limitations

This thesis uses the State Separating Proofs (SSP) methodology. In SSP, a cryptographic game is sliced into packages which provides a nice graphical representation of the relations between the cryptographic elements of a protocol.

However, as diagrams become large and complex there remains a need to refer to small but meaningful subsets of packages. As of yet, there is no language or notation to do this effectively, making references to graphs less precise. Following [BDLF+20], a middle-ground was adopted where reductions are defined as grey areas in graphs.

While avoiding the introduction of notation of every meaningful subgraph in this

work, it might be that notation for subgraphs is an inherent limitation of the SSP methodology and there is no solution yet. The proof of security of TreeKEM (Chapter 6) illustrates the problem posed by this limitation. In this proof the modelling choices divergence from the reader's expectation by ignoring nodes and focussing on layers instead. This modelling choice is difficult to explain without the ability to accurately refer to arbitrary subgraphs.

8.1.2 Contributions

The present work contributes to the SSP methodology in three ways. First, as seen in Figure 6.2, a novel way to explain relations between graphs was used. By adding coloured squares to the background, a complex graph was split into its components parts. These are easier to understand individually than as part of the whole.

Second, as introduced in Section 2.4, the present work used different arrowheads in graphs to indicate, at a glance, the kind of oracle call. Although labelling is still used in some places, the arrowheads allowed labels to be omitted in some places. This removes clutter and allowed the reader to focus on the definition of reductions.

Lastly, the present work explored a new mechanism to ensure that no two (C)SET queries for the same handle can be made to a KEY or PKEY package. In [BDLF+18] and [BDLF+20], this property was achieved via handles with cryptographic security properties plus additional state-keeping. Instead, we use the table size the key package as a counter and, in turn, assign the counter as a handle for the key contained in the current call. The advantages and disadvantages of the different mechanisms are left to be explored. Within the context of this thesis, the mechanisms allowed for a simpler model description without drawbacks.

8.2 Future Work

The present work leaves four open questions. First, this thesis provides security statements about TreeKEM and MLS's key schedule, leaving open the task of proving the security of the entire MLS protocol in an analogous corruption model, see Section 8.2.1. Second, the adversarial model can be strengthened to allow for adversaries that can adaptively corrupt key material, see Section 8.2.2. Third, the proof needs to co-evolve with the development of the RFC, see Section 8.2.3. Finally, the ad-hoc security definition for TreeKEM in this thesis can be turned into a generic primitive that constitutes as a natural building block for a continuous asynchronous group key-exchange, see Section 8.2.4.

8.2.1 Security Analysis of the Entire MLS Protocol

The scope of the present work is limited to TreeKEM and the key schedule of MLS. The security of TreeKEM and key schedule can be used as building blocks for a proof covering the entire MLS protocol. The security proofs in the present work establish that the MLS key schedule produces "good" keys in the static corruption model. In MLS those keys are used for encryption and Message Authentication Code (MAC) tags.

Starting from the guarantee of *good* keys, it should be possible to show that messages exchanged in MLS are confidential and authenticated to Group Members. From this, it follows 1) that an adversary cannot insert malicious Proposals into Groups, and 2) that, in combination with th MAC tags included in Commits, an adversary cannot insert malicious Commits into Groups.

All of the above can be proven for epochs where there is no corrupt Client in the Group. Moreover, these properties should also hold if in a previous or subsequent epoch there is a corrupt Group Member. Such notions can be formulated in a model where both long-term and ephemeral keys can be statically corrupted and can be seen as weak variants of Forward-Secrecy and Post-Compromise Security, as formulated in [KPT01].

Weak Forward-Secrecy may be proven by establishing that keys of earlier epochs look pseudorandom to new (adversarially controlled) Members. We expect that the TreeKEM security notion of the present work allows proving this in the case where the earlier epoch had only honest Members, in which case the update secret is replaced by an independent and uniformly random value. Weak Post-Compromise Security may be proven using the fact that the value at the root of the Group's TreeKEM tree is pseudorandom for a removed Member because of the blanked nodes along that Member's direct path.

However, it will be non-trivial to model the entire MLS protocol in SSP. This will be the major difficulty of continuing the proof presented in the present work. First, the scale of the MLS protocol and the ratcheting in multiple places will require large and complex graphs. This may be reduced by appropriately abstracting MLS components as bigger packages in a similar fashion to TreeKEM in this thesis. Second, the RFC's requirement for the ordering of messages will need to be enforced. This may be possible by filtering queries.

Also, it may be desirable to avoid Group "forks", where the Group is split into two or more disjoint subsets that can no longer communicate because their Group states are different. A network adversary can cause a fork by having two Members create two valid but different Commits for the same epoch and subsequently sending those two commits to two disjoint subsets of the Group's Members. Avoiding forks may also be possible by filtering queries.

8.2.2 Security Analysis Under Adaptive Corruption

The scope of the proof in the present work is limited to a static adversary. To prove that MLS provides Forward-Secrecy (FS) and Post-Compromise Security (PCS), an analysis in a stronger security model is required. Namely, it would require allowing the adversary to learn previously secret values in a TreeKEM tree, or some of a Group's keys. Then, it needs to be shown that even with this additional information the adversary cannot distinguish between honest and randomly sampled keys of fresh epochs.

It is expected that the FS and PCS guarantees of TreeKEM carry over to the MLS protocol at large. This must be verified by analysing the entire protocol under adaptive corruption.

8.2.3 Co-Evolving Proof and Protocol Standard

This thesis focusses on draft 8 of the MLS RFC. Since the release of draft 8 on Nov 14, 2019 significant changes have been made to the RFC leading up to draft 9. Relevant to this thesis, the key schedule has been changed significantly, now outputting more keys and having the option to inject a pre-shared key. The latter will require a re-examination of the key derivation procedures and updates to the security model to reflect that certain guarantees on the output of the key schedule are expected if the pre-shared key is honest.

8.2.4 TreeKEM Security Notion

Lastly, the present work uses an ad-hoc security notion in the security proof of TreeKEM (Chapter 6). However, TreeKEM is *just* a primitive that allows a group of clients to continuously and asynchronously derive key material. Therefore, one possible direction of future research is to define a static security notion for a generic continuous asynchronous group key-exchange primitive based on the security notion presented here.

Acknowledgements

I want to thank K. Kohbrok and C. Brzuska for their continuous feedback on the proof and writing and I want to thank C. Brzuska and D. Unruh for their supervision. I also want to thank B. Beurdouche, R. Barnes, and R. Robert for their responses to my questions regarding the RFC.

All diagrams found in this thesis were created with the online tool diagrams.net.

Bibliography

[ACC+] Joel Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Keep the dirt: Tainted treekem, an efficient and provably secure continuous group key agreement protocol.

[ACDT19] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the ietf mls standard for group messaging. 2019.

[An01] Jee Hea An. Authenticated encryption in the public-key setting: Security notions and analyses. Cryptology ePrint Archive, Report 2001/079, 2001. `https://eprint.iacr.org/2001/079`.

[BBM+19] Richard Barnes, Benjamin Beurdouche, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. Message layer security rfc draft 08. 2019.

[BBR18] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. *T*reeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). PhD thesis, Inria Paris, 2018.

[BD94] Mike Burmester and Yvo Desmedt. A secure and efficient conference key distribution system. In *W*orkshop on the Theory and Application of of Cryptographic Techniques, pages 275–286. Springer, 1994.

[BDLF+18] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In *I*nternational Conference on the Theory and Application of Cryptology and Information Security, pages 222–249. Springer, 2018.

[BDLF+20] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. (*Unpublished*) key-schedule security for the tls 1.3 standard. 2020.

[BL13] Daniel J Bernstein and Tanja Lange. Non-uniform cracks in the concrete: the power of free precomputation. In *I*nternational Conference on the Theory and Application of Cryptology and Information Security, pages 321–340. Springer, 2013.

[BR93] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *A*nnual international cryptology conference, pages 232–249. Springer, 1993.

[CGCD+17] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *2*017 IEEE European Symposium on Security and Privacy (EuroS&P), pages 451–466. IEEE, 2017.

[CGCG16] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *2*016 IEEE 29th Computer Security Foundations Symposium (CSF), pages 164–178. IEEE, 2016.

[CGCG+18] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In *P*roceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pages 1802–1819, 2018.

[CHK19] Cas Cremers, Britta Hale, and Konrad Kohbrok. Efficient post-compromise security beyond one group. 2019.

[dSGFW19] Cyprien Delpech de Saint Guilhem, Marc Fischlin, and Bogdan Warinschi. Authentication in key-exchange: Definitions, relations and composition. 2019.

[KMVOV96] Jonathan Katz, Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *H*andbook of applied cryptography. CRC press, 1996.

[KPT01] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Communication-efficient group key agreement. In *I*FIP International Information Security Conference, pages 229–244. Springer, 2001.

[Kra10] Hugo Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. In *A*nnual Cryptology Conference, pages 631–648. Springer, 2010.

[Lip20] Benjamin Lipp. An analysis of hybrid public key encryption. Cryptology ePrint Archive, Report 2020/243, 2020.

[PM16] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. *G*itHub wiki, 2016.

[Rog06] Phillip Rogaway. Formalizing human ignorance. In *I*nternational Conference on Cryptology in Vietnam, pages 211–228. Springer, 2006.

[SSDW88] David G Steer, Leo Strawczynski, Whitfield Diffie, and M Wiener. A secure audio teleconference system. In *C*onference on the Theory and Application of Cryptography, pages 520–528. Springer, 1988.

[Wei19] Matthew Weidner. *G*roup messaging for secure asynchronous collaboration. PhD thesis, MPhil Dissertation, 2019. Advisors: A. Beresford and M. Kleppmann, 2019-..., 2019.

[WGL00] Chung Kei Wong, Mohamed Gouda, and Simon S Lam. Secure group communications using key graphs. *I*EEE/ACM transactions on networking, 8(1):16–30, 2000.

Appendix A

Abbreviations and Acronyms

Table A.1 presents a list of acronyms used in this work. Table A.2 presents a list of abbreviations used in this work.

Table A.1: All acronyms used in the present work, alphabetically ordered.

Acronym	**Full**
AAD	**A**dditional **A**uthenticated **D**ata
AEAD	**A**uthenticated **E**ncryption with **A**ssociated **D**ata
ART	**A**synchronous **R**atched **T**ree
CAGKA	**C**ontinuous **A**synchronous **G**roup **K**ey **A**greement
CCA	**C**hosen **C**iphertext **A**ttack
DKP	**D**erive **K**ey **P**air
DPRF	**D**ual **P**seudo**r**andom **F**unction
FS	**F**orward **S**ecrecy
HPKE	**H**ybrid **P**ublic-**K**ey **E**ncryption
IETF	**I**nternet **E**ngineering **T**ask **F**orce
KDF	**K**ey **D**erivation **F**unction
KEM	**K**ey **E**ncapsulation **M**ethod
MAC	**M**essage **A**uthenticated **C**ode
MLS	**M**essaging **L**ayer **S**ecurity
PCS	**P**ost-**C**ompromise **S**ecurity
PKE	**P**ublic-**K**ey **E**ncryption
PRF	**P**seudo**r**andom **F**unction
RFC	**R**equest **F**or **C**omments
SSP	**S**tate **S**eparating **P**roofs

Table A.2: All abbreviations used in the present work, alphabetically ordered.

Abbreviation	**Full**
e.g.	**E**xempli **G**ratia (or *For Example*)
etc.	**Etc**etera
i.e.	**I**d **E**st (or *In Other Words*)
resp.	**Resp**ectively
s.t.	**S**uch **T**hat
w.r.t.	**W**ith **R**egards **T**o

Appendix B

Supplementary Proofs

The following lemma proves the existence of a reduction from $\text{GKDF2}_{\kappa}^{b,n,\lambda_s,\lambda_t}$ to $\text{GKDF}_{\kappa}^{b,n,\lambda_s,\lambda_t}$ as claimed in Section 5.2.

Lemma B.0.1. If $\exists\mathcal{A}$ that can distinguish $\text{GKDF2}_{\kappa}^{b,n,\lambda_s,\lambda_t}$, then $\exists\mathcal{R}$ s.t. $(\mathcal{A}\circ\mathcal{R})$ can distinguishing $\text{GKDF}_{\kappa}^{b,n,\lambda_s,\lambda_t}$ such that

$$\varepsilon^{\kappa}_{\text{KDF2}}(\mathcal{A}) = \varepsilon^{\kappa}_{\text{KDF}}(\mathcal{A}\circ\mathcal{R}).$$

Proof Let $\mathcal{A}$ be an adversary against $\text{GKDF2}_{\kappa}^{b,n,\lambda_s,\lambda_t}$. Then, let $\mathcal{R}$ be defined as the grey area in Figure B.1. Here, $\mathcal{R}$ intercepts queries from $\mathcal{A}$ to $\text{KEY}^{1,\lambda}$, forwarding SET and CSET queries and storing honesty bits and dishonest keys. If $\mathcal{A}$ queries dishonest keys using CGET, $\mathcal{R}$ returns the stored key for the provided handle.

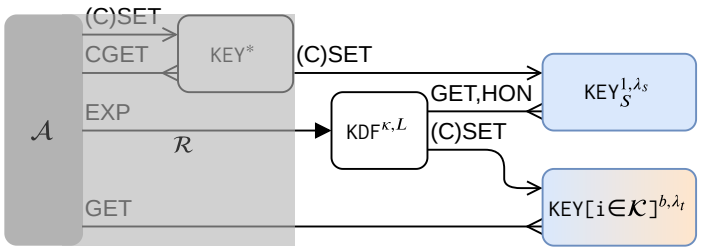


Figure B.1: Reduction adversary against $\text{GKDF}_{\kappa}^{b,n,\lambda_s,\lambda_t}$ from an adversary against $\text{GKDF2}_{\kappa}^{b,n,\lambda_s,\lambda_t}$. The KEY* package is defined in Figure B.2.

Inlining $\text{KEY}_S^{1,\lambda}$ into KEY*, as shown in Figure B.3, shows that

$$\mathcal{A}\circ\text{GKDF2}_{\kappa}^{b,n,\lambda_s,\lambda_t} \overset{\text{code}}{\equiv} (\mathcal{A}\circ\mathcal{R})\circ\text{GKDF}_{\kappa}^{b,n,\lambda_s,\lambda_t},$$

proving Lemma B.0.1.

□

Package Parameters	Package State
-	K : **table**$[h \mapsto \mathsf{k}]$ H : **table**$[h \mapsto \{0, 1\}]$

(a) The parameters and shared state of the KEY* package.

KEY*.SET(k)	KEY*.CSET(k)	KEY*.CGET(h)
$h \leftarrow$ KEY.SET(k) H$[h] \leftarrow 1$ **return** h	$h \leftarrow$ KEY.CSET(k) K$[h] \leftarrow$ k H$[h] \leftarrow 0$ **return** h	**assert** [H$[h] = 0$] **return** K$[h]$

(b) The KEY* package query implementations.

Figure B.2: Definition of the KEY* package.

KEY^*.SET(k)
assert $[\lvert\mathsf{k}\rvert = \lambda]$
$h \leftarrow \lvert\mathrm{K}\rvert$
$\mathrm{K}[h] \leftarrow_\$ \{0,1\}^\lambda$
$\mathrm{H}[h] \leftarrow 1$
$\mathrm{H}[h] \leftarrow 1$
return h

(a) Inlining KEY^1_S.SET(k) into KEY^*.

KEY^*.SET(k)
assert $[\lvert\mathsf{k}\rvert = \lambda]$
$h \leftarrow \lvert\mathrm{K}\rvert$
$\mathrm{K}[h] \leftarrow_\$ \{0,1\}^\lambda$
$\mathrm{H}[h] \leftarrow 1$
return h

(b) Removing duplicate lines of code.

$\mathsf{KEY}^{1,\lambda}$.SET(k)
assert $[\lvert\mathsf{k}\rvert = \lambda]$
$h \leftarrow \lvert\mathrm{K}\rvert$
$\mathrm{K}[h] \leftarrow_\$ \{0,1\}^\lambda$
$\mathrm{H}[h] \leftarrow 1$
return h

(c) Original KEY^1_S.SET(k) code from Figure 2.3.

KEY^*.CSET(k)
assert $[\lvert\mathsf{k}\rvert = \lambda]$
$h \leftarrow \lvert\mathrm{K}\rvert$
$\mathrm{K}[h] \leftarrow \mathsf{k}$
$\mathrm{H}[h] \leftarrow 0$
$\mathrm{K}[h] \leftarrow \mathsf{k}$
$\mathrm{H}[h] \leftarrow 0$
return h

(d) Inlining KEY_S.CSET(k) into KEY^*.

KEY^*.CSET(k)
assert $[\lvert\mathsf{k}\rvert = \lambda]$
$h \leftarrow \lvert\mathrm{K}\rvert$
$\mathrm{K}[h] \leftarrow \mathsf{k}$
$\mathrm{H}[h] \leftarrow 0$
return h

(e) Removing duplicate lines of code.

$\mathsf{KEY}^{1,\lambda}$.CSET(k)
assert $[\lvert\mathsf{k}\rvert = \lambda]$
$h \leftarrow \lvert\mathrm{K}\rvert$
$\mathrm{K}[h] \leftarrow \mathsf{k}$
$\mathrm{H}[h] \leftarrow 0$
return h

(f) Original KEY^1_S.CSET(k) code from Figure 2.3.

KEY^*.CGET(h)
assert $[\mathrm{H}[h] = 0]$
return $\mathrm{K}[h]$

(g) Original KEY^*.CGET(h) code from Figure B.2.

$\mathsf{KEY}^{1,\lambda}$.CGET(h)
assert $[\mathrm{H}[h] = 0]$
return $\mathrm{K}[h]$

(h) Original $\mathsf{KEY}^{1,\lambda}$.CGET(h) code from Figure 2.3.

Figure B.3: Inlining of $\mathsf{KEY}^{1,\lambda}_S$ into KEY^*.

The following lemma proves that the notions of *left-or-right indistinguishability under chosen-ciphertext* and *real-or-random indistinguishability under chosen-ciphertext* for PKE have equivalent distinguishing bounds, as claimed in Section 5.3.

Lemma B.0.2. If $\exists\mathcal{A}$ that can distinguish $\text{RI-IND-CCA}^b_\zeta$, then $\mathcal{R}$, defined in Algorithm 9, composed with $\mathcal{A}$ as $(\mathcal{A} \circ \mathcal{R})$ can distinguish $\text{LR-IND-CCA}^b_\zeta$ with the same advantage, i.e.

$$\varepsilon(\mathcal{A}; \text{RI-IND-CCA}^0_\zeta, \text{RI-IND-CCA}^1_\zeta) = \varepsilon(\mathcal{A} \circ \mathcal{R}; \text{LR-IND-CCA}^0_\zeta, \text{LR-IND-CCA}^1_\zeta).$$

Proof Recall the definition of $\mathcal{LR}$ from Section 5.3. Given $\mathcal{A}$ that is successful in RI-IND-CCA, let $\mathcal{R}$ be defined as Algorithm 9. $\mathcal{R}$ works by converting the left-or-right encryption oracle into a real-or-ideal oracle through hardcoding the right input to an all-zero string. Since the decryption oracle is equal between RI-IND-CCA and LR-IND-CCA, $\mathcal{R}$ exposes the decryption oracle directly to $\mathcal{A}$.

Algorithm 9 $\mathcal{R}^{e(\cdot,\cdot),d(\cdot)}(\mathsf{pk})$

$b' \leftarrow \mathcal{A}^{e(\cdot,0^\lambda),d(\cdot)}(\mathsf{pk})$
return b'

As a result, if b in LR-IND-CCA is 0, $e(\alpha, \beta)$ will return the encryption of α, which is the value provided by $\mathcal{A}$. As a result, $\mathcal{A}$ always receives the real encryption. Similarly, if b in LR-IND-CCA is 1, $e(\alpha, \beta)$ will return the encryption β, which is always the all-zero string. As a result, $\mathcal{A}$ always receives the ideal encryption.

Thus, $\mathcal{R}$'s simulation is perfect. Hence, if $\mathcal{A}$ is ε-bounded in RI-IND-CCA, $(\mathcal{A} \circ \mathcal{R})$ is ε-bounded in LR-IND-CCA.

□

The following lemma proves that an HPKE has an equivalent distinguishing probability for any adversary $\mathcal{A}$ as the underlying PKE, as claimed in Section 5.4.

Lemma B.0.3. If $\exists\mathcal{A}$ that can distinguish $\text{GHPKE}_{\zeta}^{b,\lambda}$ (Figure 5.10), then $\mathcal{R}$, defined as the grey area in Figure B.4, composed with $\mathcal{A}$ as $(\mathcal{A} \circ \mathcal{R})$ can distinguish GPKE_{ζ}^{b} (Figure 5.8) with the same advantage, i.e.

$$\varepsilon^{\zeta}_{\text{HPKE}}(\mathcal{A}) = \varepsilon^{\zeta}_{\text{PKE}}(\mathcal{A} \circ \mathcal{R}).$$

Proof Let $\mathcal{A}$ be an adversary against $\text{GHPKE}_{\zeta}^{b,\lambda}$. Then, let $\mathcal{R}$ be defined as the grey area in Figure B.4. Here, $\mathcal{R}$ simulates the behaviour of a KEY package and implements a variant of both the HPKE package (defined in Figure B.5) and PKEY (defined in Figure B.6) for $\mathcal{A}$ to interact with. As a result, the $\texttt{in}(\mathcal{A})$ interface is connected to KEY, HPKE^{*}, and PKEY^{*} from $\mathcal{R}$.

Intuitively, $\mathcal{R}$ stores values in KEY as instructed by $\mathcal{A}$. $\mathcal{R}$ encrypts those values when HPKE^{*} is invoked by retrieving the value for the specified handle from KEY, and providing that value to the PKE package.

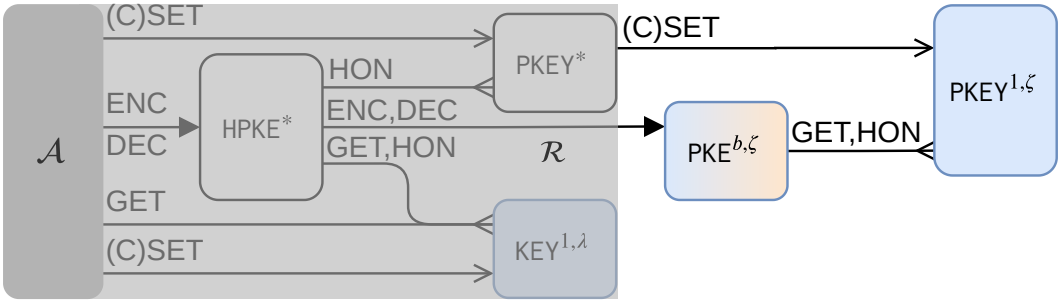


Figure B.4: Reduction against GPKE_{ζ}^{b} from an adversary against $\text{GHPKE}_{\zeta}^{b,\lambda}$. HPKE^{*} is defined in Figure B.5 and PKEY^{*} is defined in Figure B.6.

Package Parameters	Package State
-	R : **table**$[\mathsf{k} \mapsto h]$

$\mathsf{HPKE}^*.\mathrm{ENC}(h_{\mathsf{pk}}, h_{\mathsf{k}})$	$\mathsf{HPKE}^*.\mathrm{DEC}(h_{\mathsf{pk}}, c)$
assert $\mathsf{PKEY}.\mathrm{HON}(h_{\mathsf{pk}}) \vee \neg\mathsf{KEY}.\mathrm{HON}(h_{\mathsf{k}})$ $\mathsf{k} \leftarrow \mathsf{KEY}.\mathrm{GET}(h_{\mathsf{k}})$ $c \leftarrow\!\!\$\ \mathsf{PKE}.\mathrm{ENC}(h_{\mathsf{pk}}, \mathsf{k})$ $\mathrm{R}[\mathsf{k}] \leftarrow h_{\mathsf{k}}$ **return** c	$m \leftarrow\!\!\$\ \mathsf{PKE}.\mathrm{DEC}(h_{\mathsf{pk}}, c)$ **return** $\mathrm{R}[m]$

Figure B.5: Definition of a HPKE^* package.

Package Parameters	Package State
-	H : **table**$[h \mapsto \{0,1\}]$

$\mathsf{PKEY}^*.\mathrm{SET}(\mathsf{sk}, \mathsf{pk})$	$\mathsf{PKEY}^*.\mathrm{CSET}(\mathsf{sk}, \mathsf{pk})$
$(\mathsf{pk}, h) \leftarrow\!\!\$\ \mathsf{PKEY}^{b,\zeta}.\mathrm{SET}(\mathsf{sk}, \mathsf{pk})$ $\mathrm{H}[h] \leftarrow 1$ **return** (pk, h)	$(\mathsf{pk}, h) \leftarrow \mathsf{PKEY}^{b,\zeta}.\mathrm{CSET}(\mathsf{sk}, \mathsf{pk})$ $\mathrm{H}[h] \leftarrow 0$ **return** (pk, h)

Figure B.6: Definition of a PKEY^* package. The remaining queries (GET_PK, GET_SK, and HON) are defined to be equivalent to the respective queries of $\mathsf{PKEY}^{1,\zeta}$.

First, observe that the PKEY^* package simply forwards the SET and CSET queries and tracks the key honesty in the same way as the queries of $\mathsf{PKEY}^{1,\zeta}$. Therefore, inlining $\mathsf{PKEY}^{1,\zeta}$ into the SET and CSET queries will result in queries that are code equivalent to the respective queries in $\mathsf{PKEY}^{1,\zeta}$. Since the other queries are defined as the respective queries of $\mathsf{PKEY}^{1,\zeta}$, to holds that

$$(\mathsf{PKEY}^* \circ \mathsf{PKEY}^{1,\zeta}) \overset{\text{code}}{\equiv} \mathsf{PKEY}^{1,\zeta}.$$

Second, the $\mathsf{PKE}^{b,\zeta}$ package (Figure 5.9) is inlined into the HPKE^* package. The inlining result can be found in Figure B.7. A comparison of Figure 5.11 and Figure B.7 shows that

$$(\mathsf{HPKE}^* \circ \mathsf{PKE}^{0,\zeta}) \overset{\text{code}}{\equiv} \mathsf{HPKE}^{0,\zeta,\lambda}, \text{ and}$$
$$(\mathsf{HPKE}^* \circ \mathsf{PKE}^{1,\zeta}) \overset{\text{code}}{\equiv} \mathsf{HPKE}^{1,\zeta,\lambda}.$$

Therefore, $\mathcal{A} \circ (\mathcal{R} \circ \text{GPKE}^{b}_{\zeta}) \overset{\text{code}}{\equiv} \mathcal{A} \circ \text{GHPKE}^{b,\lambda}_{\zeta}$, proving Lemma B.0.3.

□

$\mathsf{HPKE}^*.\text{ENC}(h_{\mathsf{pk}}, h_{\mathsf{k}})$
assert $\mathsf{PKEY}.\text{HON}(h_{\mathsf{pk}}) \vee \neg\mathsf{KEY}.\text{HON}(h_{\mathsf{k}})$
$\mathsf{k} \leftarrow \mathsf{KEY}.\text{GET}(h_{\mathsf{k}})$
$\mathsf{pk} \leftarrow \mathsf{PKEY}.\text{GET\_PK}(h_{\mathsf{pk}})$
$c \leftarrow\!\!\$\ \zeta.\text{enc}(\mathsf{pk}, \mathsf{k})$
$\text{R}[\mathsf{k}] \leftarrow h_{\mathsf{k}}$
return c

$\mathsf{HPKE}^*.\text{DEC}(h_{\mathsf{pk}}, c)$
$\mathsf{sk} \leftarrow \mathsf{PKEY}.\text{GET\_SK}(h_{\mathsf{pk}})$
$m \leftarrow \zeta.\text{dec}(\mathsf{sk}, c)$
if $[m = \perp]$ **then**
return $\perp$
end if
return $\text{R}[m]$

(a) $\mathsf{PKE}^{0,\zeta}$ inlined into HPKE^*.

$\mathsf{HPKE}^*.\text{ENC}(h_{\mathsf{pk}}, h_{\mathsf{k}})$
assert $\mathsf{PKEY}.\text{HON}(h_{\mathsf{pk}}) \vee \neg\mathsf{KEY}.\text{HON}(h_{\mathsf{k}})$
$\mathsf{pk} \leftarrow \mathsf{PKEY}.\text{GET\_PK}(h_{\mathsf{pk}})$
if $\mathsf{PKEY}.\text{HON}(h_{\mathsf{pk}})$ **then**
$\mathsf{k} \leftarrow h_{\mathsf{k}}$
$c \leftarrow\!\!\$\ \zeta.\text{enc}(\mathsf{pk}, 0^{\lambda})$
$\text{M}[c] \leftarrow h_{\mathsf{k}}$
else
$\mathsf{k} \leftarrow \mathsf{KEY}.\text{CGET}(h_{\mathsf{k}})$
$c \leftarrow\!\!\$\ \zeta.\text{enc}(\mathsf{pk}, \mathsf{k})$
end if
$\text{R}[\mathsf{k}] \leftarrow h_{\mathsf{k}}$
return c

$\mathsf{HPKE}^*.\text{DEC}(h_{\mathsf{pk}}, c)$
$\mathsf{sk} \leftarrow \mathsf{PKEY}.\text{GET\_SK}(h_{\mathsf{pk}})$
$m \leftarrow \zeta.\text{dec}(\mathsf{sk}, c)$
if $[m = \perp]$ **then**
return $\perp$
end if
if $\mathsf{PKEY}.\text{HON}(h_{\mathsf{pk}})$ **then**
$m \leftarrow \text{M}[c]$
end if
return $\text{R}[m]$

(b) $\mathsf{PKE}^{1,\zeta}$ inlined into HPKE^*. The pseudocode of $\mathsf{HPKE}^*.\text{ENC}(h_{\mathsf{pk}}, h_{\mathsf{k}})$ is motivated by the Figure B.8.

Figure B.7: Inlining of $\mathsf{PKE}^{0,\zeta}$ into HPKE^* for $b \in \{0, 1\}$.

$\mathsf{HPKE}^*.\mathrm{ENC}(h_{\mathsf{pk}}, h_{\mathsf{k}})$
assert ...
$\mathsf{k} \leftarrow \mathsf{KEY}.\mathrm{GET}(h_{\mathsf{k}})$
$\mathsf{pk} \leftarrow \mathsf{PKEY}.\mathrm{GET\_PK}(h_{\mathsf{pk}})$
if $\mathsf{PKEY}.\mathrm{HON}(h_{\mathsf{pk}})$ **then**
$c \leftarrow\!\!\$\ \zeta.\mathrm{enc}(\mathsf{pk}, 0^{\lvert\mathsf{k}\rvert})$
$\mathrm{M}[c] \leftarrow \mathsf{k}$
else
$c \leftarrow\!\!\$\ \zeta.\mathrm{enc}(\mathsf{pk}, \mathsf{k})$
end if
$\mathrm{R}[\mathsf{k}] \leftarrow h_{\mathsf{k}}$
return c

(a) The pseudocode of $\mathsf{PKE}^{1,\zeta,\lambda}.\mathrm{ENC}(\cdot,\cdot)$ inlined into $\mathsf{HPKE}^*.\mathrm{ENC}$. The **assert** statement is truncated for brevity.

$\mathsf{HPKE}^*.\mathrm{ENC}(h_{\mathsf{pk}}, h_{\mathsf{k}})$
assert ...
$\mathsf{k} \leftarrow \mathsf{KEY}.\mathrm{GET}(h_{\mathsf{k}})$
$\mathsf{pk} \leftarrow \mathsf{PKEY}.\mathrm{GET\_PK}(h_{\mathsf{pk}})$
if $\mathsf{PKEY}.\mathrm{HON}(h_{\mathsf{pk}})$ **then**
$\mathsf{k} \leftarrow h_{\mathsf{k}}$
$c \leftarrow\!\!\$\ \zeta.\mathrm{enc}(\mathsf{pk}, 0^{\lambda})$
$\mathrm{M}[c] \leftarrow h_{\mathsf{k}}$
else
$c \leftarrow\!\!\$\ \zeta.\mathrm{enc}(\mathsf{pk}, \mathsf{k})$
end if
$\mathrm{R}[\mathsf{k}] \leftarrow h_{\mathsf{k}}$
return c

(b) Rewrite the honest pk branch to store the key handle in M. Note that $|\mathsf{k}| = \lambda$ by the use of $\mathsf{KEY}^{1,\lambda}$.

$\mathsf{HPKE}^*.\mathrm{ENC}(h_{\mathsf{pk}}, h_{\mathsf{k}})$
assert ...
$\mathsf{pk} \leftarrow \mathsf{PKEY}.\mathrm{GET\_PK}(h_{\mathsf{pk}})$
if $\mathsf{PKEY}.\mathrm{HON}(h_{\mathsf{pk}})$ **then**
$\mathsf{k} \leftarrow h_{\mathsf{k}}$
$c \leftarrow\!\!\$\ \zeta.\mathrm{enc}(\mathsf{pk}, 0^{\lambda})$
$\mathrm{M}[c] \leftarrow h_{\mathsf{k}}$
else
$\mathsf{k} \leftarrow \mathsf{KEY}.\mathrm{GET}(h_{\mathsf{k}})$
$c \leftarrow\!\!\$\ \zeta.\mathrm{enc}(\mathsf{pk}, \mathsf{k})$
end if
$\mathrm{R}[\mathsf{k}] \leftarrow h_{\mathsf{k}}$
return c

(c) Move KEY.GET to the dishonest pk branch, as it is not used in the other branch.

$\mathsf{HPKE}^*.\mathrm{ENC}(h_{\mathsf{pk}}, h_{\mathsf{k}})$
assert ...
$\mathsf{pk} \leftarrow \mathsf{PKEY}.\mathrm{GET\_PK}(h_{\mathsf{pk}})$
if $\mathsf{PKEY}.\mathrm{HON}(h_{\mathsf{pk}})$ **then**
$\mathsf{k} \leftarrow h_{\mathsf{k}}$
$c \leftarrow\!\!\$\ \zeta.\mathrm{enc}(\mathsf{pk}, 0^{\lambda})$
$\mathrm{M}[c] \leftarrow h_{\mathsf{k}}$
else
$\mathsf{k} \leftarrow \mathsf{KEY}.\mathrm{CGET}(h_{\mathsf{k}})$
$c \leftarrow\!\!\$\ \zeta.\mathrm{enc}(\mathsf{pk}, \mathsf{k})$
end if
$\mathrm{R}[\mathsf{k}] \leftarrow h_{\mathsf{k}}$
return c

(d) Replace KEY.GET by KEY.CGET as the **assert** statement ensures k is dishonest.

Figure B.8: Rewriting of $\mathsf{PKE}^{1,\zeta,\lambda}.\mathrm{ENC}(\cdot,\cdot)$ inlined into $\mathsf{HPKE}^*.\mathrm{ENC}$.

The following lemma proves that a DKP with an ideal KEY as input is statistically indistinguishable from an ideal PKEY package, as claimed in Section 5.4.

Lemma B.0.4. The key pairs resulting from a DKP construction, Figure B.9, is statistically indistinguishable from the key pairs resulting from an ideal PKEY package (see Section 2.5.2).

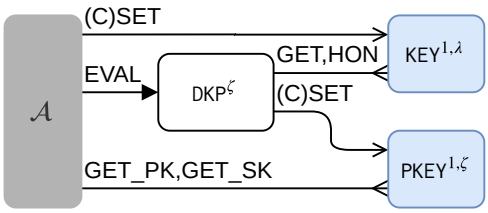


Figure B.9: The DKP construction with an ideal KEY as input, as seen in Section 5.4 and used in MLS.

Proof The proof of this Lemma B.0.4 consists of inlining the code of $\mathsf{KEY}^{1,\lambda}$ and $\mathsf{PKEY}^{0,\zeta}$ into DKP^{ζ} and find that the result is code equivalent. Figure B.10 shows the inlining process for honest seeding values. The result of which is code equivalent to $\mathsf{PKEY}^{1,\zeta}.\mathrm{SET}(\mathsf{sk},\mathsf{pk})$. Figure B.11 shows the inlining process for dishonest seeding values.

The result of the inlining shows that the behaviour of $\mathsf{DKP}^{\zeta}.\mathrm{EVAL}(\mathsf{k})$ for honest values from the KEY package is (almost) equivalent to the ideal behaviour of PKEY.SET. Similarly, the result of the inlining shows that the behaviour of $\mathsf{DKP}^{\zeta}.\mathrm{EVAL}(\mathsf{k})$ for dishonest values from the KEY package is (almost) equivalent to the ideal (and real) behaviour of PKEY.CSET. Because the distribution of seeding values of ζ may differ from the uniformly random distribution of values in $\mathsf{KEY}^{1,\lambda}$, the behaviour may be statistically different. Proving Lemma B.0.4.

□

$\mathsf{DKP}^{\zeta}.\mathrm{EVAL}(h_s)$
$s \leftarrow \mathsf{KEY}.\mathrm{GET}(h_s)$ $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \zeta.\mathrm{gen}(s)$ **return** $\mathsf{PKEY}.\mathrm{SET}(\mathsf{sk}, \mathsf{pk})$

(a) Omit code for the case that $[\mathsf{KEY}.\mathrm{HON}(h_s) = 0]$.

$\mathsf{DKP}^{\zeta}.\mathrm{EVAL}(\mathsf{k})$
$s \leftarrow\!\!\$\ \{0,1\}^{\lambda}$ $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \zeta.\mathrm{gen}(s)$ **return** $\mathsf{PKEY}.\mathrm{SET}(\mathsf{sk}, \mathsf{pk})$

(b) Inline code of $\mathsf{KEY}^{1,\lambda}.\mathrm{GET}(h)$ and $\mathsf{KEY}^{1,\lambda}.\mathrm{SET}(\mathsf{k})$.

$\mathsf{DKP}^{\zeta}.\mathrm{EVAL}(\mathsf{k})$
$s \leftarrow\!\!\$\ \{0,1\}^{\lambda}$ $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \zeta.\mathrm{gen}(s)$ $h \leftarrow \lvert\mathrm{K}\rvert$ $\mathrm{K}[h] \leftarrow (\mathsf{sk}, \mathsf{pk})$ $\mathrm{H}[h] \leftarrow 1$ **return** (pk, h)

(c) Inline $\mathsf{PKEY}^{0,\zeta}.\mathrm{SET}(\mathsf{sk}, \mathsf{pk})$.

$\mathsf{DKP}^{\zeta}.\mathrm{EVAL}(\mathsf{k})$
$s \leftarrow\!\!\$\ \{0,1\}^{\lambda}$ $h \leftarrow \lvert\mathrm{K}\rvert$ $\mathrm{K}[h] \leftarrow\!\!\$\ \zeta.\mathrm{gen}(s)$ $\mathrm{H}[h] \leftarrow 1$ **return** $(\mathrm{K}[h].\mathsf{pk}, h)$

(d) Move $\zeta.\mathrm{gen}(s)$ down.

$\mathsf{DKP}^{\zeta}.\mathrm{EVAL}(\_, \_)$
$h \leftarrow \lvert\mathrm{K}\rvert$ $\mathrm{K}[h] \leftarrow \zeta.\mathrm{gen}()$ $\mathrm{H}[h] \leftarrow 1$ **return** $(\mathrm{K}[h].\mathsf{pk}, h)$

(e) Use implicit seed for the $\zeta.\mathrm{gen}()$ algorithm.

$\mathsf{PKEY}^{1,\zeta}.\mathrm{SET}(\_, \_)$.
$h \leftarrow \lvert\mathrm{K}\rvert$ $\mathrm{K}[h] \leftarrow \zeta.\mathrm{gen}()$ $\mathrm{H}[h] \leftarrow 1$ **return** $(\mathrm{K}[h].\mathsf{pk}, h)$

(f) Original $\mathsf{PKEY}^{1,\zeta}$ code for the SET query.

Figure B.10: Inlining of $\mathsf{KEY}^{1,\lambda}$ and $\mathsf{PKEY}^{0,\zeta}$ into DKP^{ζ} for $[\mathsf{KEY}.\mathrm{HON}(h_s) = 1]$.

$\mathsf{DKP}^{\zeta}.\mathrm{EVAL}(h_s)$
$s \leftarrow \mathsf{KEY}.\mathrm{GET}(h_s)$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \zeta.\mathrm{gen}(s)$
return $\mathsf{PKEY}.\mathrm{CSET}(\mathsf{sk}, \mathsf{pk})$

(a) Omit code for the case that $[\mathsf{KEY}.\mathrm{HON}(h_s) = 1]$.

$\mathsf{DKP}^{\zeta}.\mathrm{EVAL}(\mathsf{k})$
$s \leftarrow\!\!\$\ \mathsf{k}$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \zeta.\mathrm{gen}(s)$
return $\mathsf{PKEY}.\mathrm{CSET}(\mathsf{sk}, \mathsf{pk})$

(b) Inline code of $\mathsf{KEY}^{1,\lambda}.\mathrm{GET}(h)$ and $\mathsf{KEY}^{1,\lambda}.\mathrm{CSET}(\mathsf{k})$.

$\mathsf{DKP}^{\zeta}.\mathrm{EVAL}(\mathsf{k})$
$s \leftarrow\!\!\$\ \{0,1\}^{\lambda}$
$(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \zeta.\mathrm{gen}(s)$
$h \leftarrow \lvert\mathrm{K}\rvert$
$\mathrm{K}[h] \leftarrow (\mathsf{sk}, \mathsf{pk})$
$\mathrm{H}[h] \leftarrow 0$
return (pk, h)

(c) Inline $\mathsf{PKEY}^{0,\zeta}.\mathrm{CSET}(\mathsf{sk}, \mathsf{pk})$.

$\mathsf{DKP}^{\zeta}.\mathrm{EVAL}(\mathsf{sk}, \mathsf{pk})$
$h \leftarrow \lvert\mathrm{K}\rvert$
$\mathrm{K}[h] \leftarrow (\mathsf{sk}, \mathsf{pk})$
$\mathrm{H}[h] \leftarrow 0$
return (pk, h)

(d) Move $\zeta.\mathrm{gen}(s)$ out, since $\mathcal{A}$ knows the seed it can generate the key itself.

$\mathsf{DKP}^{\zeta}.\mathrm{EVAL}(\mathsf{sk}, \mathsf{pk})$
$h \leftarrow \lvert\mathrm{K}\rvert$
$\mathrm{K}[h] \leftarrow (\mathsf{sk}, \mathsf{pk})$
$\mathrm{H}[h] \leftarrow 0$
return (pk, h)

(e) Use implicit seed for the $\zeta.\mathrm{gen}()$ algorithm.

$\mathsf{PKEY}^{b,\zeta}.\mathrm{CSET}(\mathsf{sk}, \mathsf{pk})$
$h \leftarrow \lvert\mathrm{K}\rvert$
$\mathrm{K}[h] \leftarrow (\mathsf{sk}, \mathsf{pk})$
$\mathrm{H}[h] \leftarrow 0$
return (pk, h)

(f) Original $\mathsf{PKEY}^{1,\zeta}$ code for the CSET query.

Figure B.11: Inlining of $\mathsf{KEY}^{1,\lambda}$ and $\mathsf{PKEY}^{0,\zeta}$ into DKP^{ζ} for $[\mathsf{KEY}.\mathrm{HON}(h_s) = 0]$.

Appendix C

MLS Implementation Continued

This appendix provides the pseudocode of the algorithms of MLS as a Continuous Asynchronous Group Key Agreement (CAGKA) protocol that were not present in Section 4.3.

Key Generation Figure C.1 shows the implementation of helper functions related to generating key pairs for Clients.

Objects Figure C.2 shows the implementations of Π.NewClient, Π.NewGroup, as well as Π.CreatePreSharedKey for MLS.

Proposals Figure C.3 shows the implementations of the algorithms related to sending and receiving Proposals for MLS. The RFC allows a single message to contain multiple Proposals. For simplicity, the present work models one Proposal per message. This approach has also been adopted in the changes leading up to draft 9 (see git commit `f52047a`). Relatedly, Figure C.4 and Figure C.5 show the implementation of helper functions related to proposals.

Encoding and Decoding Messages Figure C.6 and Figure C.7 show he implementation of helper functions to encode and decode messages.

Π^*.GenLLkeys()	Π^*.GenSLkeys()
$(\mathsf{sk}_{\mathrm{ll}}, \mathsf{pk}_{\mathrm{ll}}) \leftarrow \sigma.\mathrm{gen}()$ **return** $(\mathsf{sk}_{\mathrm{ll}}, \mathsf{pk}_{\mathrm{ll}})$	$(\mathsf{sk}_{\mathrm{sl}}, \mathsf{pk}_{\mathrm{sl}}) \leftarrow \zeta.\mathrm{gen}()$ **return** $(\mathsf{sk}_{\mathrm{sl}}, \mathsf{pk}_{\mathrm{sl}})$

Figure C.1

Π.NewGroup($\mathcal{C}$)

$\{\mathsf{sk}_{\text{ll}}, \mathsf{pk}_{\text{ll}}, \ldots\} \leftarrow \mathcal{C}$
$(\mathsf{sk}_{\text{sl}}, \mathsf{pk}_{\text{sl}}) \leftarrow \Pi^*.\text{GenSLkeys}()$

$\text{id} \leftarrow\!\!\$\ \{0,1\}^8$
$cth \leftarrow 0$
$ith \leftarrow 0$
$\texttt{is} \leftarrow 0$
$e \leftarrow 0$
$ln \leftarrow \{\mathcal{C}.cc, \mathsf{pk}_{\text{sl}}\}$ ▷ `LeafNode`
$rt \leftarrow [ln]$
$th \leftarrow 0$
$gc \leftarrow \{\text{id}, e, th, cth\}$ ▷ `GroupContext`

$\mathcal{S} \leftarrow \{gc, rt, ith\}$
$\mathcal{S}.\text{keys} \leftarrow \{\mathsf{sk}_{\text{ll}}, \mathsf{pk}_{\text{ll}}, \mathsf{sk}_{\text{sl}}, \mathsf{pk}_{\text{sl}}, \texttt{is}\}$
$\mathcal{S}.\text{proposals} \leftarrow [\,]$
return $\mathcal{S}$

Π.NewClient()

$\text{id} \leftarrow\!\!\$\ \{0,1\}^{32}$
$(\mathsf{sk}_{\text{ll}}, \mathsf{pk}_{\text{ll}}) \leftarrow\!\!\$\ \Pi^*.\text{GenLLkeys}()$
$cc \leftarrow \Pi^*.\text{CreateCredentials}(\mathsf{pk}_{\text{ll}})$
return $\{\text{id}, \mathsf{sk}_{\text{ll}}, \mathsf{pk}_{\text{ll}}, cc\}$

Π.CreatePreSharedKey($\mathcal{C}$)

$(\mathsf{sk}_{\text{sl}}, \mathsf{pk}_{\text{sl}}) \leftarrow \Pi^*.\text{GenSLkeys}()$

$\text{kid} \leftarrow\!\!\$\ \{0,1\}^8$
$s \leftarrow \sigma.\text{sig}(\mathcal{C}.\mathsf{sk}_{\text{ll}}, \{\text{kid}, cs, \mathsf{pk}_{\text{sl}}, \mathcal{C}.cc\})$

$\mathcal{C}.psk \leftarrow\!\!/\!\!/\ (\mathsf{sk}_{\text{sl}}, \mathsf{pk}_{\text{sl}})$

$cik \leftarrow \{\text{id}, cs, \mathsf{pk}_{\text{sl}}, cc, s\}$
return $(\mathcal{C}, cik)$

Figure C.2

Π.ProposeAdd($\mathcal{S}, cik$)

$p \leftarrow \{cik\}$ ▷ Add
$(\mathcal{S}, ct) \leftarrow\!\!\$\ \Pi^*$.EncodeMessage$(\mathcal{S}, 2, p)$
$\mathcal{S} \leftarrow \Pi$.ReceiveProposal$(\mathcal{S}, ct)$
return $(\mathcal{S}, ct)$

Π.ProposeRemove($\mathcal{S}, \mathsf{pk}_{\mathrm{sl}}$)

$i \leftarrow$ index of `LeafNode` with $\mathsf{pk}_{\mathrm{sl}}$
if $[i = \perp]$ **then return** $(\diamond, \diamond)$

$p \leftarrow \{i\}$ ▷ Remove
$(\mathcal{S}, ct) \leftarrow\!\!\$\ \Pi^*$.EncodeMessage$(\mathcal{S}, 2, p)$
$\mathcal{S} \leftarrow \Pi$.ReceiveProposal$(\mathcal{S}, ct)$
return $(\mathcal{S}, ct)$

Π.ProposeUpdate($\mathcal{S}$)

$(\mathsf{sk}_{\mathrm{sl}}, \mathsf{pk}_{\mathrm{sl}}) \leftarrow \Pi^*$.GenSLkeys()

$p \leftarrow \{\mathsf{pk}_{\mathrm{sl}}\}$ ▷ Update
$(\mathcal{S}, ct) \leftarrow\!\!\$\ \Pi^*$.EncodeMessage$(\mathcal{S}, 2, p)$
$\mathcal{S} \leftarrow \Pi$.ReceiveProposal$(\mathcal{S}, ct)$

$\mathcal{S}$.keys $\leftarrow\!\!/\!/ (\mathsf{sk}_{\mathrm{sl}}, \mathsf{pk}_{\mathrm{sl}})$
return $(\mathcal{S}, ct)$

Π.ReceiveProposal($\mathcal{S}, ct$)

$pt \leftarrow \Pi^*$.DecodeMessage$(\mathcal{S}, ct)$
If $[pt = \diamond]$ **return** $\diamond$

$\{p, \ldots\} \leftarrow pt$
If invalid(p) **return** $\diamond$

$\mathcal{S}$.proposals $\leftarrow\!\!/\!/\ p$
return $\mathcal{S}$

Figure C.3

Π^*.ProposalsToList(ps)

$as, rs, us, is \leftarrow [\,]$
for $p \in ps$ **do**
  $\{i_{\text{sender}}, \ldots\} \leftarrow p$
  pid $\leftarrow \{i_{\text{sender}}, \mathrm{H}(p)\}$ ▷ ProposalID
  add pid to the appropriate list
end for
return $us \| rs \| as$

Figure C.4

Π^*.ApplyProposals(rt, ps, pids)

for pid $\in$ pids **do**
  $p \leftarrow$ Proposal with pid in ps
  if $[p = \perp]$ **then**
    return $\diamond$
  else if p is update **then**
    $\{sender, leaf\_key, \ldots\} \leftarrow p$
    $node \leftarrow$ leaf node at index $sender$
    $node.\mathsf{pk} \leftarrow leaf\_key$
  else if p is remove **then**
    $\{removed, \ldots\} \leftarrow p$
    $node \leftarrow$ leaf node at index $removed$
    while $[node \neq root]$ **do**
      $blank(node)$
      $node \leftarrow parent(node)$
    end while
  else if p is add **then**
    $\{init\_key, cc, \ldots\} \leftarrow p$
    $node \leftarrow init\_key$
    extend rt if necessary
    $i \leftarrow$ left-most empty leaf node
    add i to $unmerged\_leafs$ in each node to root
    while $[node \neq root]$ **do**
      $blank(node)$
      $node \leftarrow node.parent$
    end while
    $rt[i] \leftarrow \{init\_key, cc\}$
  end if
end for
return rt

Figure C.5

Π^*.EncodeMessage($\mathcal{S}, content\_type, content$)

$\{gc, \text{gid}, e, rt, \mathsf{sk}_{\text{ll}}, \mathsf{pk}_{\text{sl}}, \ldots\} \leftarrow \mathcal{S}$
$i_{\text{sender}} \leftarrow \Pi^*.\text{GetSenderIndex}(rt, \mathsf{pk}_{\text{sl}})$

$\{sds, \ldots\} \leftarrow \mathcal{S}.\text{keys}$
if $[content\_type = 3] \vee [content\_type = 2]$ **then**
    $\{\mathsf{hk}, \ldots\} \leftarrow \mathcal{S}.\text{keys}$
    $g \leftarrow \mathcal{S}.g_{hk}$
    $\mathsf{k} \leftarrow \Pi^*.\text{ExpandLabel}(\mathsf{hk}, \texttt{"hs key"}, i_{\text{sender}}, \Pi^*.\text{KeyLen}, gc)$
    $n \leftarrow \Pi^*.\text{ExpandLabel}(\mathsf{hk}, \texttt{"hs nonce"}, i_{\text{sender}}, \Pi^*.\text{NonceLen}, gc) \oplus g$
else
    omitted, beyond the scope of the current work
    return $\diamond$
end if

$sdn \leftarrow\!\!\$\ \{0,1\}^8$
$sd \leftarrow \{i_{\text{sender}}, g\}$ ▷ MLSSenderData
$\alpha_{sd} \leftarrow \{\text{gid}, e, content\_type, sdn\}$ ▷ MLSCiphertextSenderDataAAD
$sd_{\text{enc}} \leftarrow\!\!\$\ \xi.\text{enc}(sds, sdn, sd, \alpha_{sd})$

$si \leftarrow \{\mathcal{S}.\text{context}, \text{gid}, e, i_{\text{sender}}, content\_type, content\}$ ▷ SignatureInput
$s \leftarrow \sigma.\text{sig}(\mathsf{sk}_{\text{ll}}, si)$
$pt \leftarrow \{\text{gid}, e, i_{\text{sender}}, content\_type, content, s\}$ ▷ MLSPlaintext

$ctc \leftarrow \{pt, \sigma\}$ ▷ MLSCiphertextContent
$\alpha_{ctc} \leftarrow \{\text{gid}, e, content\_type, sdn, sd_{\text{enc}}\}$ ▷ MLSCiphertextContentAAD
$pt_{\text{enc}} \leftarrow\!\!\$\ \xi.\text{enc}(\mathsf{k}, n, ctc, \alpha_{ctc})$
$ct \leftarrow \{\text{gid}, e, content\_type, sdn, \alpha_{sd}, sd_{\text{enc}}, pt_{\text{enc}}\}$ ▷ MLSCiphertext

$\mathcal{S}.g_{hk} \leftarrow (\mathcal{S}.g_{hk} + 1)$
return $(\mathcal{S}, ct)$

Figure C.6

Π^*.DecodeMessage$(\mathcal{S}, ct)$

$\{gc, \ldots\} \leftarrow \mathcal{S}$
$\{\text{gid}, e, content\_type, sdn, \alpha_{sd}, sd_{\text{enc}}, pt_{\text{enc}}\} \leftarrow ct$ ▷ `MLSCiphertext`

$\{sds, \ldots\} \leftarrow \mathcal{S}$.keys
$sd \leftarrow \xi$.dec$(sds, sdn, sd_{\text{enc}}, \alpha_{sd})$
if $[sd = \perp]$ **return** $\diamond$
$\{i_{\text{sender}}, g\} \leftarrow sd$

if $[content\_type = 3] \vee [content\_type = 2]$ **then**
  $\{\mathsf{hk}, \ldots\} \leftarrow \mathcal{S}$.keys
  $\mathsf{k} \leftarrow \Pi^*$.ExpandLabel$(\mathsf{hk}, \texttt{"hs key"}, i_{\text{sender}}, \Pi^*.\text{KeyLen}, gc)$
  $n \leftarrow \Pi^*$.ExpandLabel$(\mathsf{hk}, \texttt{"hs nonce"}, i_{\text{sender}}, \Pi^*.\text{NonceLen}, gc) \oplus g$
else
  omitted, beyond the scope of the current work
  return $\diamond$
end if

$\alpha_{ctc} \leftarrow \{\text{gid}, e, content\_type, sdn, sd_{\text{enc}}\}$ ▷ `MLSCiphertextContentAAD`
$pt \leftarrow \xi$.dec$(\mathsf{k}, 0, pt_{\text{enc}}, \alpha_{ctc})$ ▷ `MLSPlaintext`
if $[pt = \perp]$ **return** $\diamond$

$\{\text{gid}, e, i_{\text{sender}}, content\_type, content, s\} \leftarrow pt$
$\mathsf{pk}_{\text{ll}} \leftarrow$ long-lived public key in $\mathcal{S}$.rt at index i_{sender}
$si \leftarrow \{\mathcal{S}.\text{context}, \text{gid}, e, i_{\text{sender}}, content\_type, content\}$ ▷ `SignatureInput`
if $\neg\sigma$.ver$(\mathsf{pk}_{\text{ll}}, si, s)$ **return** $\diamond$

return pt

Figure C.7

Appendix D

Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Eric Cornelissen**,
*(*author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Cryptographic Analysis of the Message Layer Security Protocol in the Static Corruption Model,
   *(*title of thesis)

   supervised by Chris Brzuska, Dominique Unruh, and Konrad Kohbrok.
   *(*supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Eric Cornelissen
15/05/2020