Discreet: distributed delivery service with context-aware cooperation

Ludovic Paillat, Claudia-Lavinia Ignat, Davide Frey, Mathieu Turuani, Amine Ismail

DiSCreet: Distributed Delivery Service with Context-Aware Cooperation

Ludovic Paillat[1,2*], Claudia-Lavinia Ignat[2], Davide Frey[3], Mathieu Turuani[2], Amine Ismail[1]

[1]Hive Computing Services, Cannes, France.
[2]Université de Lorraine, CNRS, Inria, LORIA, Nancy, France.
[3]Inria, IRISA, CNRS, Université de Rennes, Rennes, France.

*Corresponding author(s). E-mail(s): ludovic.paillat@hivenet.com;
Contributing authors: claudia.ignat@inria.fr; davide.frey@inria.fr;
mathieu.turuani@inria.fr; amine.ismail@hivenet.com;

Abstract

End-to-end encrypted messaging applications such as Signal became widely popular thanks to their capability to ensure the confidentiality and integrity of online communication. While the highest security guarantees were long reserved to two-party communication, solutions for n-party communication remained either inefficient or less secure until the standardization of the MLS Protocol (Messaging Layer Security). This new protocol offers an efficient way to provide end-to-end secure communication with the same guarantees originally offered by the Signal Protocol for two-party communication. However, both solutions still rely on a centralized component for message delivery, called the Delivery Service in the MLS Protocol. The centralization of the Delivery Service makes it an ideal target for attackers and threatens the availability of any protocol relying on MLS. In order to overcome this issue, we propose DiSCreet (Distributed delIvery Service with Context-awaRE coopEraTion), a design that allows clients to exchange protocol messages efficiently and without any intermediary. It uses a Probabilistic Reliable-Broadcast mechanism to efficiently deliver messages and the Cascade Consensus Protocol to handle messages requiring an agreement. Our solution strengthens the availability of the MLS Protocol without compromising its security. We compare the theoretical performance of DiSCreet with another distributed solution, the DCGKA protocol, and detail the implementation of our solution.

Keywords: Distributed systems, Group key agreement, Consensus protocols, Reliable broadcast

1 Introduction

To protect the privacy of their users, a number of Internet-based services have started to develop solutions based on *end-to-end encryption (E2EE)* that prevent third parties from accessing user data transferred from one endpoint to another.

Secure messaging applications such as Signal and WhatsApp are well-known to advertise their use of *E2EE*. Indeed, the Signal Protocol was the first to propose *E2EE* for two-party conversations using the Double Ratchet algorithm [26]. However, the solutions proposed by these same messaging applications for n-party secure communication

were either inefficient by requiring the establishment of encrypted communication between all pairs of group members, or less secure when using a common encryption key (e.g. Sender Keys Protocol [5]) which may not be refreshed as the group dynamically changes.

To address the topic of secure group communication, industrial and academic organizations such as Cisco, Mozilla, Facebook and Inria proposed the Messaging Layer Security Protocol (MLS) standardized as RFC 9420 [6]. This protocol relies on a Group Key Agreement Protocol called TreeKEM [9] allowing members of a group to derive a common secret called *group key* which serves as a basis to secure group communications. It is scalable in terms of the number of operations modifying the group, and it supports periodic group-key renewals preventing compromised communication.

The MLS Protocol offers an efficient solution to guarantee the *confidentiality* and *integrity* of communication. However, the *availability* of the protocol depends on the *Delivery-Service* component, which remains centralized most of the time. The centralization of this component makes it an ideal target for attackers who wish to disrupt communication. Notably, with the help of a compromised Delivery Service, an attacker can prevent group members from refreshing their keys and resolving the compromise.

In order to overcome these limitations we propose DiSCreet (Distributed delIvery Service with Context-awaRE coopEraTion), a solution to establish a distributed Delivery Service. It combines two distributed communication mechanisms adapted to the need of the messages exchanged by the protocol. We use a Probabilistic Reliable Broadcast mechanism [18] to reliably deliver messages allowing users to propose changes to the group (i.e. *Proposal* messages) and the Cascade Consensus Protocol [1] to deliver the messages that actually modify the group (i.e. *Commit* messages) and thus require an agreement between members.

Then, we evaluate the theoretical performance of DiSCreet and compare it with another distributed protocol: DCGKA [31] (Decentralized Continuous Group Key Agreement). Our analysis shows that our approach promises good performance, notably in the context of dynamic groups whose users may join or leave at any time.

Our contribution is three-fold:

- the formalization of the MLS protocol's Delivery Service, detailing the necessary properties of this component;
- a novel algorithm describing DiSCreet, a fully distributed Delivery Service, completed by an implementation based on an open-source MLS implementation;
- a theoretical study comparing the complexity of our approach with other existing distributed solutions.

We start by reviewing the state of the art of distributed communication mechanisms in Section 2. We then present the TreeKEM Protocol and formalize the Delivery Service in Section 3. Section 4 details the communication protocols chosen to build our solution and unfolds the execution of our algorithm. Section 5 presents a theoretical study comparing the complexity of our solution with the DCGKA protocol [31]. Section 6 discusses the implementation of our solution. Finally, Section 7 concludes the paper and presents some future work directions.

This article is an extended version of our previous work [25] published in the 16th International Symposium on Foundations & Practice of Security. In this revision, we expand the State of the Art to discuss new Group Key Agreement protocols and detail existing distributed communication mechanisms. Additionally, we present the result of a theoretical study evaluation our solution, as well as a presentation of the implementation of our solution.

2 Related work

Secure communication requires securely sharing encryption keys thus preventing attackers from gaining access to them. Additionally, as communications can remain established for a long time, protocols must provide a way to mitigate the eventual compromise of some keys (e.g. a computer is infected by a malware, an attacker gets physical access to a device, ...). The following security properties need to be ensured:

- *Forward secrecy* (FS): the knowledge of a key does not affect the security of previous keys.
- *Post-Compromise security* (PCS): if a member is compromised, an UPDATE operation from this

member will resolve the compromise and restore the security of the group in subsequent states.

Figure 1 summarizes the scope of these two properties in the event of a member's compromise.

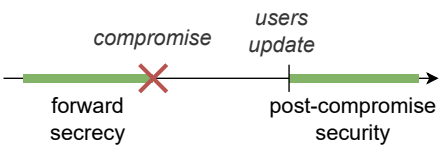


Fig. 1: Illustration of the protection given by Forward secrecy and Post-Compromise security properties against a compromise.

The Signal protocol based on the Double Ratchet Algorithm [26] ensures these two security properties for two-party communication. The Double Ratchet algorithm provides *Forward secrecy* by generating a new encryption key for each message while periodical Diffie-Hellman key exchanges provide fresh security material ensuring *Post-Compromise security*. However, this approach cannot be applied to group (i.e. n-party) communication and the use of pairwise communication channels (i.e. a group member sends a message to the group by using $n-1$ secure communication channels established with the other members) does not scale well.

An alternative, *Sender Keys* [5], allows one member to use pairwise communication channels to share a common encryption key that can be used to encrypt messages. While this key can be used to deterministically generate individual *message keys* to provide *Forward Secrecy*, *Post-Compromise security* is not ensured as the common key is only renewed in rare occasions such as member removal. Thus, the compromise of one member makes it possible to spy on future messages for a long time.

Secure group-communication protocols address these drawbacks. In the following, we first present the main existing protocols for secure group communication, and we highlight the advantages of TreeKEM [9]. We then describe the distributed communication mechanisms we selected to make it distributed.

2.1 Secure Group Communication

First attempts to secure group communication were Conference Key Distribution Systems in which a member generates a conference key and distributes it to all group members. Different topologies for Distribution Systems [13] were proposed. However, the *Star-Based* topology is inefficient as it requires $\mathcal{O}(n)$ key exchanges, while other topologies such as a *Tree* or a *Cyclic System* cannot tolerate the fault of even one participant.

Group Key Agreement protocols were introduced as a way to establish and manage dynamic groups whose members can derive a common encryption key called *group key*. These protocols mainly provide three operations: the ADD and REMOVE operations to manage the group and the UPDATE operation to allow group members to refresh their secret keys. Each operation leads to a new group key, thereby ADD and REMOVE operations guarantee *backward* and *forward secrecy* for the group key while periodical UPDATEs ensure *post-compromise security*. Thus, if a member does not renew its secret key, this member should be removed from the group as a compromise of this member threatens the security of the group.

Collaborative Group Key Agreement protocols were the first to establish a *group key* built on the principle of the Diffie-Hellman key exchange. The GDH protocols [4, 28] allow members to collaboratively build the group key presented in Equation 1. However, the computation of this key requires $\mathcal{O}(n)$ communication rounds.

$$K = \alpha^{r_1 \times r_2 \times \cdots \times r_n} \tag{1}$$

The TGDH (Tree-based Group Key Agreement) protocol [19] establishes a group key more efficiently in $\mathcal{O}(\log n)$ rounds. The underlying binary-tree structure contains the member keys in the leafs, intermediate keys exchanged between subgroups (i.e. members sharing one node in their path to the root of the tree) and the root representing the group key. The resulting tree for a 5-member group is illustrated in Figure 2 and the associated group key is presented in Equation 2. Additionally, operations on the group only require the modification of $\mathcal{O}(\log n)$ intermediate keys.

$$K = \alpha^{\left(\alpha^{\left(\alpha^{r_1 \times r_2}\right) \times \left(\alpha^{r_3 \times r_4}\right)}\right) \times r_5} \tag{2}$$

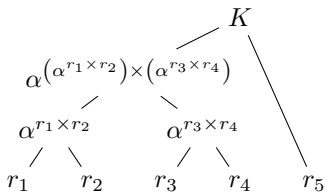


Fig. 2: Illustration of the tree built in TGDH for a 5-member group.

Nevertheless, the TGDH protocol cannot tolerate the fault or disconnection of any member, as each member can only modify their path to the root of the tree. Thus, the removal of one member from the tree requires the participation of the closest neighbor to this member in the tree. The ART (Asynchronous Ratcheting Trees) protocol [15] enhances TGDH with the capability of creating a group in which all members but the group creator are not required to be online. In this protocol, the group creator can create the initial tree by using the X3DH key exchange algorithm [23] with ephemeral keys stored by each group member in a Public Key Infrastructure. This key exchange guarantees that each group member will be able to derive only the keys they should know except for the group creator who initially knows the entire tree.

The more recent TreeKEM protocol [9] allows clients to issue asynchronous group operations. By replacing the tree structure based on Diffie-Hellman with a tree structure based on the principle of *Key Derivation*, the TreeKEM protocol allows any member to carry out operations on the tree without requiring the help of any particular group member. The TreeKEM protocol achieves good performance and was backed by multiple security analyses (i.e. [2], [12], ...). It is part of a complete solution for Secure Group Communication, the MLS Protocol [6], standardized in RFC 9420.

The Ratcheting Tree structure of the protocol, which will be presented in Section 3.2, allows group operations to be performed using a logarithmic number of key encryptions and key derivations compared to the size of the group.

MLS combines TreeKEM with TreeSync [29], a protocol providing a method to authenticate the content of the tree structure in TreeKEM. TreeSync introduces data structures such as a Merkle tree in which members are required to sign every modification of the tree. Therefore, any member can verify the integrity of the tree (i.e. ensure that nodes were only modified by authorized members), otherwise a member could perform a *double-join attack* by modifying a key that there are not allowed to know. Then, with the knowledge of this key, the attacker could spy on the group even after having been removed, as the modified key might not be immediately replaced.

As TreeSync does not assume anything about the content of the key tree, and is well separated from the group key agreement protocol, our solution, DiSCreet, can integrate TreeSync and thus benefits from its security features.

2.2 Protocol handling concurrent operations

In distributed systems, the case of concurrent operations (i.e. operations from multiple members that appear to take place at the same time) is expected and unavoidable. Thus, protocols must provide a way to handle concurrent operations and, if necessary, reconcile conflicting ones. However, the method chosen to handle concurrent operations, can highly impact the performance, or the confidence given to a central server.

In TreeKEM, the main challenge would be to concurrently modify the underlying tree containing keys that are shared by multiple users. The chosen solution, which will be described in details in Section 3.1, is to allow concurrent operations in the form of *Proposals* and then have one member apply these operations in the tree and resolve conflicts in the form of *Commits*. However, one issue is that doing so the member creating the commit would modify keys that they are not allowed to know, and then perform attacks such as *double-join*. TreeKEM solves this issue by introducing the concept of *blank nodes*: when the member creating the commit need to modify keys they are not allowed to know, this member simply replace these keys by empty nodes that do not contain any secrets. Figure 3 presents a scenario involving a blank node when member C updates their secret k_C to k'_C concurrently with member A updating from k_A to k'_A. In this scenario, the member A is the one to commit. Therefore, member A replaces two keys in the tree (i.e. k_1 and k_2 are replaced by k'_1 and k'_2) in order to update the group key

and introduces a blank node to insert member C's update.

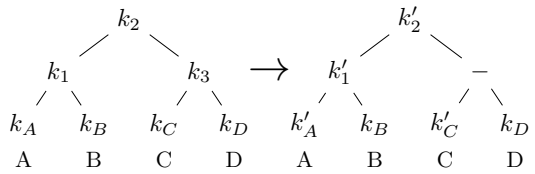


Fig. 3: Illustration of the introduction of blank nodes when committing another user's proposal. Here member C updates concurrently with member A and member A is the one committing the operations.

The use of blank nodes makes it possible to guarantee TreeKEM's tree invariant: a member only knows the keys on its path to the root of the key. Nevertheless, using blank nodes leads to a non-binary tree, as the children of a blank node are virtually attached to the first non-blank parent. Thus, the use of blank nodes can affect the performance of TreeKEM as multiple group member might not share an encryption key and require individual encryptions instead.

The Tainted TreeKEM protocol [20] proposed an alternative to blank nodes called *tainted nodes*. The idea is to allow group members to modify any key in the tree when committing. However, to maintain the security, a key will be marked as *tainted* when it was modified by a member not allowed to know this key. Therefore, to safely remove a member, one also has to update all nodes that were tainted by the removed member. Doing so will guarantee that if this member maliciously kept the knowledge of those keys, this knowledge will not allow the removed member to spy on the group and perform *double-join attacks*.

A more theoretical approach [10] studied the trade-off between post-compromise security, concurrency and communications. One of the proven result is that a minimum of two communication rounds are needed to achieve PCS when supporting concurrent operations. This is indeed the case in TreeKEM, with the separation of Proposals and Commits into two different phases.

The CoCoA protocol [3] proposed a different approach to handle concurrent operations. In order to optimize communication while reconciling concurrent updates, the protocol employs a central server to safely merge updates on the shared structure: the key tree, and crafts user-tailored message to communicate only the relevant updated keys.

One of the first approaches for a fully distributed Group Key Agreement was Causal TreeKEM [30]. The protocol provides the capability of merging concurrent operations and allows distributed communication using a causally-ordered broadcast primitive. However, operations can be merged only if one provides a commutative operator to merge public-private key pairs in the key tree. Additionally, the protocol suffers from multiple security issues: one of them is that in the event of concurrent removals, every group key derived concurrently to these operations is insecure as these group keys can still be derived, if at least two removed members collaborate with each other.

Finally, the DCGKA protocol [31] (Decentralized Continuous Group Key Agreement) provides an interesting solution for securing group communications without requiring a central server. The protocol employs a principle similar to the one introduced by Sender Keys [5], with one member having its own ratchet that allow them to encrypt messages to the group. Then updates can be shared through a causal-order broadcast that removes the need for a central server to operate the protocol. Additionally, when one member proposes an update, the underlying update secret is used by all members to update their ratchet, thus achieving PCS for all group members with only one operation.

The drawback of this protocol is that its complexity increases linearly to the group size for the member issuing an operation. The underlying reason is that when a member performs an operation, this member securely exchanges a new secret with every group member, by encrypting this secret individually to each group member.

In our article, we rely on TreeKEM [6] as a basis for our proposed distributed Group Key Agreement mechanism, as the protocol seems more scalable in the number of cryptographic operations and multiple implementations are available to conduct an experimental study. However, our solution also works with similar protocols such as Tainted TreeKEM [20], which was not the object of any known implementation.

2.3 Distributed Communication Mechanisms

The main challenge of creating a distributed protocol operating without a central server is to ensure consistency across the whole group. Furthermore, the protocol needs to be resilient against the failure of a certain number of group members, called *processes* in distributed systems, as well as network failures or unexpected delays. We will choose mechanisms resistant to failures adapted to the different messages of TreeKEM as described in Section 3.1.

2.3.1 Reliable Broadcast

Reliable Broadcast protocols focus on the correct delivery of one message from a given sender. These protocols ensure *Termination* which states that all correct processes eventually delivery a message sent by a correct process. Additionally, in the case of a *Byzantine* (i.e. faulty or malicious) sender, the protocols will ensure *Agreement*, meaning that all correct processes will deliver the same message.

Bracha's Algorithm [11] achieves reliable broadcast asynchronously and supports the Byzantine failure of at most t processes among $n = 3t+1$ processes. This protocol orchestrates the exchange of Echo messages for the processes to reach an *agreement*, then Ready messages allow processes to ensure *termination*.

Another approach is the High-Throughput Secure Reliable Broadcast Protocol [22] that aims at reducing the number of exchanged messages and thus increase throughput. When broadcasting messages, processes sign their messages, as well as digests of previously received messages. These digests are interpreted as acknowledgements and by tracking these acknowledgements in a graph, any process can determine when to deliver a message by counting the processes which acknowledged the message either directly or indirectly. However, the protocol is not designed to ensure low latency and targets environments with highly active users, which is not likely in group key agreement protocols.

The Scalable Reliable Broadcast protocol [18] aims at reducing the number of exchanged messages in order to improve performance and scale to larger groups. Indeed, the protocol achieves an $\mathcal{O}(\log(n))$ per-process communication and computation complexity while ensuring the reliable broadcast properties with high probability. Incrementally, 3 algorithms are introduced for scalable reliable broadcast. First, Murmur only provides *termination*, and can be seen as a broadcast protocol that only guarantees that a message gets delivered by all processes. Then, Sieve built on Murmur only guarantees *agreement*. Finally, Contagion relies on Sieve to provide both properties of reliable broadcast.

2.3.2 Consensus protocols

Contrary to Reliable Broadcast, Consensus protocols provide stronger guarantees and allows all group members to decide on one value or one message, which translates into a global order of messages across the group. A consensus protocol offers properties comparable to those of *agreement* and *termination*. However, with a consensus protocol the *agreement* will apply for messages sent by all group members.

Contrary to reliable broadcast protocols, the FLP impossibility result [16] demonstrates that in the asynchronous model, in which the execution rate of processes and communication delays are not bounded, it is impossible to guarantee *agreement* and *termination* in the event of faulty processes.

The Paxos protocol [21] achieves consensus by guaranteeing *agreement* under the condition that at most t processes are faulty among $n = 2t + 1$ processes. However, the protocol can guarantee *termination* only during synchronous periods, i.e. when processes and messages do not experience exceedingly long delays. Additionally, the protocol does not address the existence of Byzantine processes.

The PBFT protocol [14] (Practical Byzantine Fault Tolerance) achieves consensus even under the presence of Byzantines processes. This protocol is leader-based and can tolerate the Byzantine failure of at most f processes among $n \geq 3f + 1$ participants to the consensus. Like for Paxos, termination requires additional synchrony assumptions: e.g. to detect a faulty leader in the consensus protocol.

Context-Adaptative Cooperation (CAC) is a recent broadcast abstraction that sits between

reliable broadcast and consensus [1]. It allows multiple senders to send concurrent messages and focuses on the ability to detect when some messages are indeed sent concurrently and conflict with each other.

In CAC, contrary to classical reliable broadcast protocols, the protocol will not only deliver a message but also a *conflict-set* indicating other messages that might conflict with the delivered one. Therefore, CAC can act like reliable broadcast when there are no conflicts during a broadcast instance. Otherwise, in the case of a conflict between multiple senders, this conflict will be detected and CAC can trigger a classical consensus protocol to handle the conflict.

Cascade Consensus Protocol.

The CAC abstraction enables the construction of a consensus protocol named *Cascade Consensus*. In this protocol, a process starts by using CAC to broadcast a message. Then, if there are no conflicts, meaning that CAC delivers a single message with a *conflict-set* of size 1, the protocol can directly finish and deliver this message. Otherwise, in case of a conflict, with a *conflict-set* containing two messages or more, the protocol triggers a *Restrained Consensus* between the senders associated with the messages in the *conflict-set*. This *Restrained Consensus* involves only the senders in conflict and thus is less costly than regular consensus provided that none of its participants is Byzantine and no period of asynchrony occurs during its execution. Thus, either one of the participants is chosen by *Restrained Consensus* and its message can be delivered, or in the event of a timeout, a classical Consensus algorithm is used to settle the *Cascade Consensus* protocol and reach a final decision.

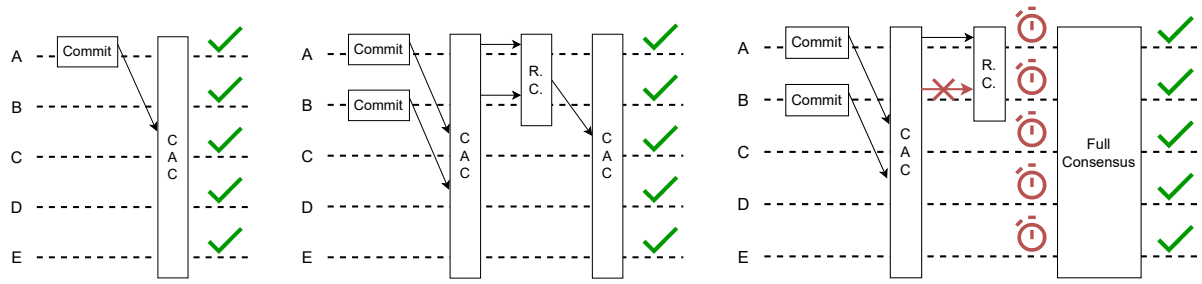
These different scenarios allowing the early termination of Cascade Consensus are represented in Figure 4. In a general case, where there is little activity and processes exchange messages with low latency, no conflicting broadcasts will occur. Therefore, the execution of Cascade Consensus will fall under the scenario Figure 4a and the protocol will terminate after one CAC broadcast. Otherwise, in some cases like the one in Figure 4b, a few processes will broadcast messages concurrently. Then, the CAC protocol will detect the conflict and result in the delivery of a *conflict-set* with multiple messages. The conflicting processes will initiate a Restrained Consensus algorithm based on this *conflict-set* and most likely reach an agreement that can be broadcast through a second CAC instance. Finally, in an unfavorable situation (e.g. Byzantine failures, network issues, ...) either the Restrained Consensus algorithm or the second CAC broadcast will fail, and a timeout will trigger a Full Consensus, as shown in Figure 4c. This Full Consensus will be a classical Consensus algorithm that will be more costly but will ensure that all processes reach an agreement.

3 The TreeKEM Protocol and its Delivery Service

The TreeKEM protocol underlying Messaging Layer Security (MLS) [6] provides a standard and well-secured solution for end-to-end secure group communication such as group messaging and video conferencing. We first present the *Propose and Commit* approach used by TreeKEM to handle concurrent operations in the group. Next, we introduce the core structure of TreeKEM based on the ratcheting tree. We then describe the Delivery Service component of TreeKEM which is in charge of handling protocol communication and resolving conflicts between members. We present our formalization of the role and the properties of this Delivery Service component. Finally, we describe the centralized solution of the Delivery Service adopted by TreeKEM and its follow-up protocols.

3.1 Propose and Commit

The 8$^{\text{th}}$ draft of the MLS Protocol, introduces the principle of *Propose and Commit* in the TreeKEM Protocol which describes the current way of organizing the operations in the protocol. This allows group members to perform concurrent operations in the form of *Proposals*. Then, these proposals can be merged into a *Commit* which materializes these changes into the group. Each new state of the group is then referred to as an *epoch*. Therefore, each *epoch* is associated to a new group key and group members should go through the same *epochs* when running the protocol. However, the protocol does not precise which members should commit some proposals, and thus it is up to the



(a) In the absence of conflicts, the consensus terminates after one CAC instance.

(b) When a conflict is detected, conflicting members settle using a Restrained Consensus algorithm and broadcast the result in a second CAC instance.

(c) In the presence of Byzantine failures or network issues that prevent the termination of Restrained Consensus, a timer will trigger a full consensus that will guarantee an agreement under degraded conditions.

Fig. 4: Different possible executions of the Cascade Consensus protocol depending on the graceful conditions that allow the protocol to terminate faster.

application using the protocol to decide which client commits the *Proposals*.

In the case of conflicting *Proposals* (e.g. multiple add/remove/update *Proposals* for the same member), the *committer* must choose between those proposals to solve the conflicts. Additionally, if one *Commit* contains *Proposals* to add one or multiple members, the protocol generates a *Welcome* message that allows them to effectively join the group.

Therefore, based on the MLS RFC [6] we propose the following formalization of the TreeKEM protocol:

- $s \leftarrow \textsc{Init}(ID)$: initiates the group state s for the user associated with the identity ID.
- $p \leftarrow \textsc{ProposeAdd}(s, ID')$: creates a proposal to add the user corresponding to ID' to the group with the current state s. The operation outputs a proposal p.
- $p \leftarrow \textsc{ProposeRemove}(s, ID')$: creates a proposal to remove the user corresponding to ID' from the group with current state s. The operation outputs a proposal p.
- $p \leftarrow \textsc{ProposeUpdate}(s, k)$: creates a proposal to update the current user's key in current state s with k. The operation outputs a proposal p.
- $(C, W) \leftarrow \textsc{Commit}(s, P)$: commits a set P of valid and non-conflicting proposals (i.e. according to Section 12.2 of the RFC [6]) to make the current group state s progress. The operation outputs a commit message C to make current members transition to the new group state and possibly a welcome message W to allow proposed new users to join the group.
- $(s', K) \leftarrow \textsc{Process}(s, M)$: processes a commit or welcome message M to transition from the old group state s to the new group state s'. The transition additionally generates a new *group key* K.

3.2 Ratcheting Tree

To implement the protocol, TreeKEM defines the structure of the ratcheting tree. The main objective of this structure is to provide scalability and support larger groups compared to Sender Keys [5] and other protocols which usually scale linearly with the size of the group.

The ratcheting tree is organized as a balanced binary tree of public-private key-pairs whose leaves represent members of the group. Group members do not share the same view of the tree as each member only knows a subset of the private keys: the keys on the path from the leaf associated with this member to the root of the tree. Therefore, the only private key known by all members is the root of the tree which gets to be used as the group key.

Figure 5 presents the example of a ratcheting tree in the case of a 4-member group in addition to an example of group operation that demonstrates the benefits conveyed by the structure. In the initial stage on the left, each member has the exclusive knowledge of their secret keys k_A, k_B, k_C

and k_D, while members A and B share the knowledge of the private key k_1, C and D the knowledge of k_3, and all are aware of the group key k_2.

When performing a group operation, for example when member A updates its secret, the member must refresh all keys on its path to the root, as illustrated in the right part of Figure 5. The refreshed keys are generated by derivation from the randomly picked leaf secret k'_A: k'_1 is derived from k'_A and k'_2 is derived from k'_1. The member also needs to securely share these keys with other members of the group. Using the ratcheting tree, the member can use the keys that were not replaced and limit the number of encryptions: k'_2 can be encrypted with the public key of k_3 which directly covers the two members C and D. Therefore, the ratcheting tree structure reduces the number of encryptions to communicate new group operations.

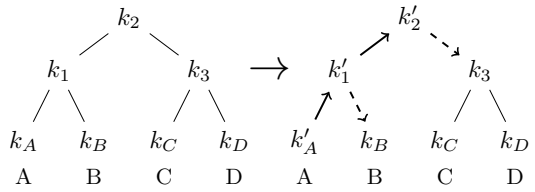


Fig. 5: Outcome of updating a path in the ratcheting tree of a 4-member group. Plain arrows indicate key derivations while dashed arrows symbolize key encryptions.

With the ratcheting tree structure and the key-derivation mechanism, the number of keys and needed encryptions to generate a Commit message is limited to $log(n)$ in a group of size n, which offers good scalability. However, as discussed in Section 2.2 the presence of blank nodes in the ratcheting tree could decrease the performance of the protocol depending on the quantity of these nodes.

However, the protocol does not provide a way to merge the modifications conveyed by multiple commits. Thus, in the event of concurrent commits, only one commit must be chosen to be applied by all group members. The task of resolving these conflicts falls under the responsibility of the *Delivery Service* that will be described in the following subsection.

3.3 The TreeKEM's Delivery Service

The MLS Architecture draft [8] describes the typical architecture in which clients implementing the MLS Protocol [6] interact with each other. In this architecture, clients communicate with each other using a *Delivery Service* whose detailed description falls out of the scope of the MLS Protocol.

Thus, the role of the *Delivery Service* is essentially to ensure the delivery of the different types of messages: Application messages exchanged between users, *Proposal* messages and *Commit* messages. The only constraint for Application and *Proposals* messages is that they must eventually reach their recipients.

However, *Commit* messages present different challenges. In fact, the MLS Protocol imposes a linear history of epochs. This means that the members should process the *Commit* messages in the same order and only one *Commit* message can exist per epoch, as the protocol is not capable of merging multiple *Commits*. Ensuring this linear history can be a problem as the members may generate concurrent commits in different situations:

- On the one hand, some members might not take advantage of concurrent operations (i.e. send a *Proposal* directly followed by a corresponding *Commit*) and in dynamic and/or large groups, there is a high probability of members issuing multiple *Commits* simultaneously.
- On the other hand, even when members first send *Proposals* to allow operations to be executed concurrently, there is still the need for one of those members to *commit* the operations. This member can be determined deterministically to limit the cost, for example based on the set of received proposals. But, in a distributed system the presence of failures and asynchronous periods may prevent all members from deciding on the same *committer*. Thus, even if the probability of conflicts is lowered, it is still possible for concurrent *commits* to be issued. Addressing this situation requires solving consensus.

Therefore, in case of conflicts, the *Delivery Service* must act as a reference, capable of deciding

on one *Commit* that all members will consider as the right one for a given *epoch*.

3.3.1 Formalization.

Based on the specifications provided in the MLS Protocol [6] and the MLS Architecture draft [8], we now present our formalization of the role and the properties of the TreeKEM Delivery Service.

The Delivery Service is a group communication mechanism that provides two operations and two callbacks:

- `ds_proposal_broadcast`(pm): a process p_i can invoke this operation to submit a proposal message pm.
- `ds_proposal_deliver`(p_i, pm): callback triggered to deliver a proposal message pm broadcast by process p_i.
- `ds_commit_propose`(ep, cm): a process p_i can invoke this operation to submit a commit message cm for the current epoch ep.
- `ds_commit_deliver`(ep, p_i, cm): callback triggered to deliver the commit message cm for the epoch ep and broadcast by process p_i, which allows members to progress to the next epoch and state.

The Delivery Service satisfies the following properties:

- **Proposal Validity.** If a correct process p_i ds-proposal-delivers a pair (p_j,pm), then pm is a valid proposal message, p_j is a correct process and p_j has previously ds-proposal-broadcast pm.
- **Proposal Totality.** If a correct process p_i ds-proposal-broadcasts a proposal message pm, then all correct processes eventually ds-proposal-deliver the pair (p_i,pm).
- **Epoch Validity.** If a correct process p_i ds-commit-delivers a tuple (ep,p_j,cm), then ep is the current epoch, cm is a valid commit message in the current epoch, p_j is a correct process and p_j has previously ds-commit-proposed cm.
- **Epoch Agreement.** If any two correct processes ds-commit-deliver respective tuples (ep,-,cm) and (ep',-,cm'), and if $ep = ep'$ then we have $cm = cm'$.
- **Epoch Termination.** If a correct process p_i ds-commit-proposes a commit message cm for the epoch ep, then all correct processes eventually ds-commit-deliver a pair (ep,-,-).
- **Epoch-Content Consistency** If a correct process ds-commit-delivers a tuple (-,-,cm), then for all proposal messages pm referenced by cm all correct processes have previously ds-proposal-delivered a tuple (-,pm).

3.4 Centralized Delivery Service

The approach adopted by TreeKEM and other protocols it inspired (e.g. [20], [3]) is to rely on a central authority to implement the Delivery Service. In this case, the Delivery Service would be operated by a central server. To keep confidentiality this central server is assumed to be *untrusted*. This means that, as the messages are end-to-end encrypted, the server cannot read or modify the messages exchanged between members, but has access to a limited set of metadata necessary to fulfill its task (i.e. group ID and epoch).

Due to its limited view on the content of messages, an *untrusted* centralized Delivery Service can only accomplish the role of an ordering server, providing the different messages in the same order for all clients. Then, in case of conflicts, the members can choose the first valid commit message and coherently resolve the conflict.

However, this central server can easily become a target of attacks such as Denial of Service which can impact its availability. In more complex settings, a compromised Delivery Service can be used to mount attacks against specific groups. By blocking the operations of given members, one can prevent these members from refreshing their keys and thus eventually lead to the failure of the Post-Compromise Security property.

4 A Distributed Delivery Service

In this section, we describe our solution for a fully distributed Delivery Service. Instead of relying on a third-party untrusted server, the participants in the key agreement protocols, or a subset of them, directly run the delivery service. We first present the communication mechanisms chosen to make the TreeKEM Delivery Service distributed. Next, we detail DiSCreet, our solution for a fully distributed Delivery Service for Group Key Agreement protocols.

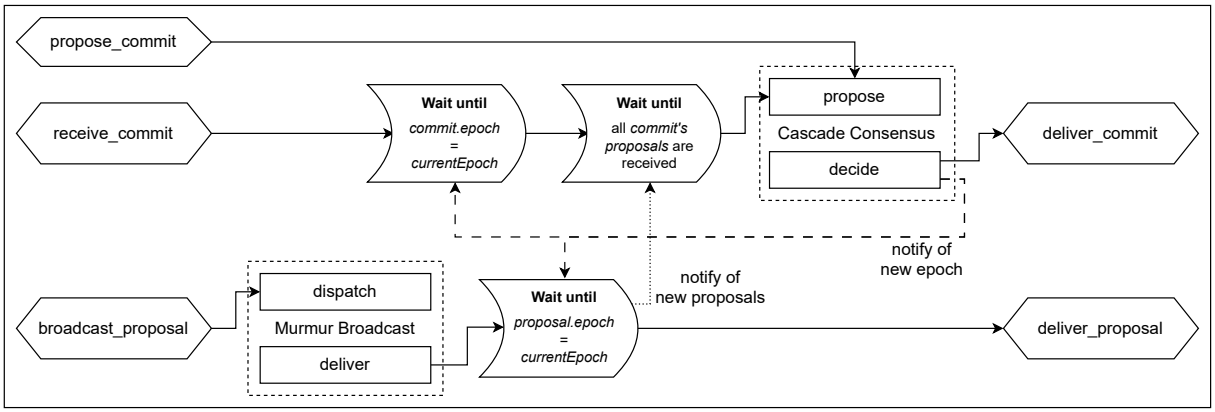


Fig. 6: Flowchart detailing DiSCreet, our solution for a Distributed Delivery Service. It illustrates the steps Proposals and Commits go through before being delivered to the higher level application and the TreeKEM protocol.

4.1 Communication mechanisms

As described Section 3.1, Proposal and Commit messages have different functions and thus different constraints. Therefore, we adopted separate communication mechanisms, tailored to the need of each message type.

In TreeKEM, proposal messages only describe operations, that will later be applied to the group via a commit. Most of the constraints, such as guaranteeing that all members keep the same view of the ratcheting tree, are passed on to the subsequent commit messages. Therefore, we only need to ensure that sent proposal messages will eventually be received by all group members.

We adopt Murmur [18], a gossip-based dissemination protocol which only guarantees the delivery of messages without the ability to guarantee their consistent ordering across recipients. This protocol perfectly fits the requirements for Proposal messages as these messages do not need to be ordered in any way.

Commit messages need to be totally ordered, with group members' agreeing on this total order. Consensus protocols satisfy this requirement.

We choose the Cascade Consensus protocol [1] as the graceful conditions allowing the early termination of Cascade Consensus match the expected behavior of TreeKEM. Indeed, depending on the size and the dynamics of a group, conflicts may not occur very often, in which cases the consensus ends after a CAC broadcast. Additionally, these conflicts may usually be restricted to a few members of the group, reducing the complexity of consensus by triggering Restrained Consensus.

4.2 Algorithm

The main idea of our solution is to combine two protocols, the Murmur protocol [18] and the Cascade Consensus protocol [1], while addressing the issues that arise from combining these protocols with the TreeKEM protocol. Our solution then ensures that messages from the two protocols are delivered in a correct order, while enforcing the property of *Epoch-Content Coherency* which is orthogonal to both protocols.

We illustrate our solution as a flowchart in Figure 6 and formally in Algorithm 1. The flowchart in Figure 6 describes the verification process and the delays that messages can encounter between their receipt from the communication component and their delivery. Algorithm 1 details how the execution unfolds.

Handling proposals

When a member wants to issue a group operation (e.g. *Add*, *Remove* or *Update*), it starts by creating a proposal with the TreeKEM protocol and submits it to the Delivery Service. The Delivery Service then takes over (l. 10) and forwards the proposal to the Murmur [18] protocol (l. 11) in charge of efficiently propagating this message to the entire group.

Upon completion, all group members receive the proposal (l. 20). When a member receives

Algorithm 1 Formalization of our solution for a Distributed Delivery Service based on the Cascade Consensus Protocol and the Murmur broadcast protocol.

Implements:

1: DistributedDeliveryService, **instance** dds

Uses:

2: MurmurBroadcast, **instance** mb
3: CascadeConsensusBroadcast, **instance** ccb
4: TreeKEMProtocol, **instance** tkem
5:
6: **upon event** < Init > **do**
7: $proposals = \emptyset$; $incomplete\_commits = \emptyset$; $future\_commits = \emptyset$;
8: $future\_proposals = \emptyset$
9:
10: **upon event** < dds.BroadcastProposal | [`Proposal`, $proposal$] > **do**
11: **trigger** < mb.Dispatch | [`Proposal`, $proposal$] >
12:
13: **procedure** *handleCommit*($commit$) ▷ handle commits that are or became complete
14: **if** $commit$.epoch > $tkem$.currentEpoch **then**
15: $future\_commits \leftarrow future\_commits \cup commit$
16: **else**
17: ccb.validateCommit($commit$)
18: **end if**
19:
20: **upon event** < mb.Deliver | [`Proposal`, $proposal$] > **do** ▷ ignore past and invalid proposals
21: $proposals \leftarrow proposals \cup proposal$
22: **for** $commit \in incomplete\_commits$ **do**
23: **if** $commit$.proposals $\subseteq proposals$ **then**
24: *handleCommit*($commit$)
25: $incomplete\_commits \leftarrow incomplete\_commits \setminus commit$
26: **end if**
27: **end for**
28: **if** $proposal$.epoch = $tkem$.currentEpoch **then**
29: **trigger** < dds.DeliverProposal | [`Proposal`, $proposal$] >
30: **else**
31: $future\_proposals \leftarrow future\_proposals \cup proposal$
32: **end if**
33:
34: **upon event** < dds.ProposeCommit | [`Commit`, $commit$] > **do**
35: **trigger** < ccb.Propose | $tkem$.currentEpoch,[`Commit`, $commit$] >
36:
37: **upon event** < ccb.ReceiveCommit | [`Commit`, $commit$] > **do** ▷ ignore past and invalid commits
38: **if** $commit$.proposals $\not\subseteq proposals$ **then**
39: $incomplete\_commits \leftarrow incomplete\_commits \cup commit$
40: **else**
41: *handleCommit*($commit$)
42: **end if**

```
43: upon event < ccb.Deliver | [Commit, commit] > do
44:     tkem.apply(commit)
45:     for proposal ∈ future_proposals do
46:         if tkem.currentEpoch = proposal.epoch and tkem.isValid(proposal) then
47:             trigger < dds.DeliverProposal | [Proposal, proposal] >
48:             future_proposals ← future_proposals \ proposal
49:         end if
50:     end for
51:     for commit ∈ future_commits do
52:         if tkem.currentEpoch = commit.epoch and tkem.isValid(commit) then
53:             ccb.receiveCommit(commit)
54:             future_commits ← future_commits \ commit
55:         end if
56:     end for
57:     trigger < dds.DeliverCommit | [Commit, commit] >
```

a proposal message, two cases can arise. If the proposal belongs to the current epoch, and thus satisfies *Proposal Validity*, we can directly deliver the message (l. 29). Otherwise, we wait for this message to become valid by inserting it in the list of future proposals (l. 31).

Figure 7 illustrates the example of a scenario in which a member has to wait for a message to become valid. In this scenario, a proposal is received before the commit it depends on. This situation may arise as the MURMUR protocol is designed to be efficient and thus may deliver messages faster than Cascade Consensus. Thus, it is imperative that our solution addresses this issue by waiting before delivering the proposal.

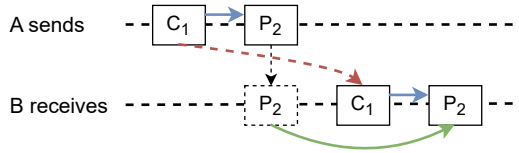


Fig. 7: Example of out of order message reception. In this circumstance, member A sends a commit C_1 and later a proposal P_2 that depends on C_1. However, member B receives the proposal before the commit. The desired behavior is to wait for the commit, and then deliver the proposal message.

We also address the opposite scenario of a commit being received before one of its proposals or more. Upon reception of a new proposal, we check if this proposal was referenced by any previously received commit (l. 22) and unblock this commit (l. 24) once we receive all of its proposals.

Handling commits

Then, to materialize the submitted operations, and modify the group according to these changes, a group member will eventually propose a commit to other group members. As this commit would modify shared structures of TreeKEM, such as the Ratcheting Tree described Section 3.2, we condition the acceptance of a commit to an agreement (i.e. the property of *Epoch Agreement*) with other group members. As described previously, the use of the Cascade Consensus protocol [1] fulfills our requirements for exchanging commit messages. Therefore, when a member wishes to propose a commit (l. 34), the Delivery Service's first step is to forward the commit to Cascade Consensus (l. 35).

The next step of the Delivery Service is to ensure that only valid commit messages are delivered. We seek to satisfy the *Epoch Validity* and *Epoch-Content Consistency* properties in order to ensure that the delivered commit corresponds to the current group status (i.e. the current epoch) and that it only introduces valid group operations respectively.

For this purpose, we introduce a slight modification to the Cascade Consensus Protocol and to the CAC broadcast mechanism on which it is based (see Section 2). Specifically, we delay the acknowledgement of a commit (i.e. the emission of a WITNESS message upon receipt of a first

WITNESS message) until the client considers this commit as valid. We materialize this validation by a call to `ccb.validateCommit` (l. 17). This ensures that the commit message is selected as a candidate in the consensus protocol only when all members agree on its validity.

Delaying commits allows our solution to control the flow of epochs, making sure that it only delivers messages belonging to the current epoch, and that all members have moved to the current epoch before deciding on the next one. This prevents an attacker from executing a *Denial of Service attack* by submitting many commits in a short time. To implement this delay, Algorithm 1 employs two waiting queues ($incomplete\_commits$, $future\_commits$) initialized on line 7. By means of these queues, we wait until all the proposals referenced by a commit are received before starting an agreement (i.e. a consensus) on this commit. This leads to the satisfaction of the *Epoch-Content-Consistency* property.

Therefore, to handle commit messages we employ the modified cascade-consensus mechanism (l. 37). Commits whose proposals are missing are delayed (l. 39). If the commit belongs to the current epoch, we have to wait for missing proposals in order to guarantee *Epoch-Content Consistency*. If the commit belongs to a future epoch (l. 15) that cannot yet be verified, waiting is necessary in order to guarantee *Commit Validity*. Then, valid commits are transmitted (l. 17) as candidates for the next epoch to the Cascade Consensus Protocol. This protocol ensures the *Epoch Agreement* and *Epoch Termination*. As previously shown in Figure 4, the protocol can deliver the commit efficiently. Indeed, in the absence of conflicts only one CAC broadcast is sufficient to deliver the commit. Otherwise, a Restrained Consensus or a Full Consensus might be triggered.

Finally, when members agree on the same commit, this commit is delivered by the consensus protocol (l. 43). This commit can be transmitted to the TreeKEM protocol to take effect. Additionally, the change of epoch can unblock proposals (l. 47) and commits (l. 53) that were previously delayed if the client received messages out of order.

5 Theoretical evaluation

We present the expected performance of our solution by means of key metrics for Group Key Agreement protocols (i.e. the complexities in terms of the number of cryptographic operations and the size of exchanged messages) and key metrics for distributed protocols (i.e. the complexity in terms of the number of messages exchanged).

We provide the expected performance of three protocols. First, we study the performance of the TreeKEM protocol [9] combined with DiSCreet. Similarly, we present the performance obtained by combining the Tainted TreeKEM protocol [20] with DiSCreet, as Tainted TreeKEM uses the same kind of messages as TreeKEM. Finally, we compare these two approaches with the DCGKA protocol [31] which is another distributed solution for secure group communication.

To highlight the benefits but also the drawbacks of our solution, we choose to compare the three solutions under two relevant scenarios:

- Non-dynamic groups: in *non-dynamic groups*, also known as *static groups*, the members can be considered static, as members are rarely added or removed from the group. Then, the group activity is mostly composed of PCS Updates, which are simply called *update* operations in the chosen protocols. This is the case for most groups in messaging applications or for teams and departments in a company.
- Dynamic groups: in *dynamic groups* the members of the groups change with high frequency, at least more often than the frequency we require for PCS Updates. The activity of those groups mostly consists of *add* and *remove* operations. These groups better describe video and audio conferences as well as volunteer/distributed computing groups in which devices can join and leave groups frequently.

These two scenarios highlight the main difference between the three protocols that we studied. Indeed, in DCGKA one PCS Update is enough to provide PCS for all group members simultaneously. However, in TreeKEM and Tainted TreeKEM, each group member has to submit a PCS Update to ensure the security of the group.

Additionally, we also consider the sub-case of using concurrent operations as complexities vary for the TreeKEM and Tainted TreeKEM protocols

in the face of concurrent operations. On the other hand, DCGKA supports concurrent operations, but these operations do not affect the complexity of the protocol (i.e. for t concurrent operations in DCGKA, the overall complexity will always be t times the complexity of one operation).

First we present the complexities of the protocols in the general case, and then we detail the complexities in the studied scenarios.

5.1 Cryptographic operations

For the studied protocols, we consider as cryptographic operations the asymmetric encryption/decryption operations, as well as the key derivations to construct the ratcheting tree. In practice, these key derivations result from the generation of an asymmetric key-pair from a given *seed*, which can be considered as a costly operation.

In TreeKEM, each operation requires emitting a proposal and then a commit. The emission of a proposal is negligible as it only requires one signature. However, the sender of a commit has performed $\log n$ key derivations and $\log n - 1$ key encryptions, as described in the Figure 5 of Section 3.2. Then, any other group member that receives this commit will decrypt 1 key and compute the remaining parent keys with at most $\log n - 1$ key derivations. Therefore, the complexities of cryptographic operations are $\mathcal{O}(\log n)$ for both the committer and the recipients.

These complexities are valid in the case of a balanced ratcheting tree. However, in the presence of concurrent operations, the tree may contain several blank nodes that deteriorate its performance. The concurrent updates, as well as the blank nodes in the tree, may require the committer to perform at most $n - 1$ key encryptions in the worst case, leading to a complexity of $\mathcal{O}(n)$ for concurrent operations and subsequent operations.

In the Tainted TreeKEM protocol, the complexities are identical when operations are sequential but slightly differ when considering concurrent operations. In the case of sequential operations the protocol does not incorporate blank nodes into the tree which prevents the increase of complexity for subsequent operations. This improvement is made despite the increased size of the commit message due to a larger number of public keys sent by the committer. However, this does not modify the complexity that is kept to $\mathcal{O}(n)$ and only reduces the complexity of subsequent operations to $\mathcal{O}(\log n)$.

In DCGKA, the initiator of an operation generates a secret update and encrypts this secret to the $n - 1$ other members individually, resulting in a complexity of $\mathcal{O}(n)$. In contrast, the recipient is required to decrypt the update secret and sign an acknowledgement. This results in a complexity of $\mathcal{O}(1)$, as updating each member's secret is done using a simple hash function whose complexity is negligible compared to key derivations in TreeKEM.

5.2 Size of messages

In TreeKEM, proposals consist of one update, thus their sizes are constant. Commit messages, however convey the information of one updated path in the tree: $\log n - 1$ public keys and $\log n - 1$ seeds allowing selected subgroups to complete their view of the tree. We also consider the size of the control messages exchanged by DiSCreet. When considering an optimistic scenario in which the network and clients act as expected, a commit is delivered after one CAC broadcast in which the largest message is the broadcast of one READY message containing around n signatures. Therefore, we obtain a complexity of $\mathcal{O}(n)$ for both the committer and each recipient messages' size.

In case of concurrent operations, the committer may send $n - 1$ encrypted seeds in the worst case where all group members issue concurrent updates or the entire tree is blanked. Additionally, in Tainted TreeKEM, the committer may send a commit updating the entire tree, thus containing n public keys. However, these cases do not modify the complexity $\mathcal{O}(n)$ for the size of the messages.

In DCGKA, the complexities for the size of messages follow the ones for the number of cryptographic operations. The initiator has to encrypt the update secret $n - 1$ times resulting in a complexity of $\mathcal{O}(n)$, while recipients broadcast one acknowledgement leading to a complexity of $\mathcal{O}(1)$.

5.3 Number of messages exchanged

In TreeKEM and similarly in Tainted TreeKEM, we have to consider the delivery of both proposals and commits.

Each proposal message is broadcast using the MURMUR protocol [18]. As explained in the article detailing the protocol, to broadcast the message,

each member will forward the message to a sample of a logarithmic number of members. This results in a complexity of $\mathcal{O}(n \log n)$ messages exchanged globally for the delivery of one proposal.

In the case of commits, the complexities are determined by the Cascade Consensus protocol [1]. If we consider the same optimistic case as in Section 5.2, a commit is delivered after the broadcast of one WITNESS message and one READY message by each member, resulting in a complexity of $\mathcal{O}(n^2)$ messages globally exchanged.

In DCGKA, both messages describing operations, and subsequent acknowledgement must be delivered using a reliable broadcast protocol. The most efficient protocol to achieve reliable broadcast while minimizing the number of exchanged messages is the CONTAGION protocol [18]. Using this protocol the broadcast of either one operation or one acknowledgement can be done with $\mathcal{O}(n \log n)$ messages exchanged. Therefore, to actually complete one group operation, with one member sending an operation and the $n - 1$ others responding with an acknowledgement, the protocol has a complexity of $\mathcal{O}(n^2 \log n)$ messages exchanged.

5.4 Complexities comparison

We first compare the different protocols in the first scenario, involving a static group performing a PCS Update of all of its members. Table 1 presents the results of this comparison.

In the case of TreeKEM and Tainted TreeKEM, we are required to perform n PCS Updates. Two options are possible, performing n sequential updates or performing all of these updates concurrently. If we perform n sequential PCS updates, the total complexity is computed by multiplying by n the different complexities. The communication's complexity is derived from the cost of the n Cascade Consensus that were required. Additionally, in that case where no blank nodes were introduced in the tree, the subsequent operations are not affected and their complexity remains logarithmic.

If we perform n concurrent PCS updates, complexities depend on whether the member submitted one concurrent operation or committed the operations. In Table 1 we refer to members who submit one operation as *Arbitrary process* whereas the committer is referred to as the *Sender process*. The complexity for commit recipients is only logarithmic, while the complexity becomes linear for the committer. Additionally, the large amount of concurrent operations degrades the performance for subsequent operations in TreeKEM compared to Tainted TreeKEM. This difference is highlighted in the column called *Subsequent operation* which presents the complexity in terms of the number of cryptographic operations that are performed by any member issuing a commit after the n sequential or concurrent operations. In the case of the communications, the cost decreases compared to performing n sequential updates. Indeed, as the n concurrent PCS updates require only one Cascade Consensus, the communication's complexity is dictated by the cost of the n reliable broadcasts of proposals.

Finally, in the case DCGKA, the complexities are simply the ones for performing one operation, as one PCS Update disseminates a secret update that can be applied to all members. When describing the complexity of cryptographic operations in Table 1, the member creating the PCS Update is referred to as the *Sender process* whereas other members are labelled *Arbitrary process*.

This first scenario, shows that DCGKA is always a better solution for static groups. However, when using TreeKEM and Tainted TreeKEM, we can achieve a similar cost for the number of communications, as well as for the cryptographic operations performed by the member with the highest workload. However, this performance can only be achieved by coordinating all members to submit concurrent updates, and it requires using Tainted TreeKEM for a reasonable performance.

Then, we choose to compare the different protocols in the second scenario, involving a dynamic group where members often perform *Add* and *Remove* operations. Table 2 presents the result of this comparison, in two circumstances: one without concurrent operations, and one with t concurrent operations.

In these settings, we define the *Sender process* as the process issuing a commit in TreeKEM protocols, and as any process issuing an operation in DCGKA. On the contrary, we consider as *Receiver process* a process that does not commit in TreeKEM protocols and a process that does not issue an operation in DCGKA.

Table 1: Comparison of the complexities for TreeKEM and Tainted TreeKEM using DiSCreet, and DCGKA, in the context of a static group, whose activity mainly consists of periodically performing PCS Updates of all its members.

Protocol	Complexity of cryptographic operations			Sent messages size for one process		Global number of exchanged messages
	Arbitrary process	Sender process[a]	Subsequent operation[b]	Arbitrary process	Sender process[a]	
TreeKEM with DiSCreet						
n subsequent updates	$\mathcal{O}(n \log n)$	–	$\mathcal{O}(\log n)$	$\mathcal{O}(n^2)$	–	$\mathcal{O}(n^3)$
n concurrent updates	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2 \log n)$
Tainted TreeKEM with DiSCreet						
n subsequent updates	$\mathcal{O}(n \log n)$	–	$\mathcal{O}(\log n)$	$\mathcal{O}(n^2)$	–	$\mathcal{O}(n^3)$
n concurrent updates	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2 \log n)$
DCGKA 1 update provide PCS for all	$\mathcal{O}(1)$	$\mathcal{O}(n)$	–	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2 \log n)$

[a]Complexity for the only member who commits in TreeKEM protocols, or issues the operation in DCGKA.
[b]Complexity of submitting one *commit* after the n updates.

In the absence of concurrent operations, protocols behave with their expected complexities. The difference is that contrary to PCS Updates, the cost of each operation matters in the DCGKA protocol. Therefore, the cost for the recipients is the lowest in the DCGKA protocol. However, the complexity for senders (i.e. committer in TreeKEM-like protocols) and for communications is lower in TreeKEM and Tainted TreeKEM compared to DCGKA.

This difference can be even more relevant in the second case, i.e. in the presence of concurrent operations. Indeed, the cost of concurrent operations increases linearly in the DCGKA protocol, whereas the cost for recipients remains unchanged for TreeKEM and Tainted TreeKEM. Only the cost for committer increases to $\mathcal{O}(t + t \log \frac{n}{t})$, as shown by [10]. This cost translates to a logarithmic number of operations when t grows significantly more slowly than n, and increases to a linear cost similar to that of DCGKA otherwise. Additionally, the cost of communications increases slower for TreeKEM and Tainted TreeKEM compared to DCGKA, as a concurrent operation only involves one additional proposal broadcast, while it requires n acknowledgements in DCGKA.

In the presence of concurrent operations, the use of Tainted TreeKEM rather than TreeKEM remains relevant, as if t gets closer to n, the subsequent commits in TreeKEM have a cost of $\mathcal{O}(n)$ cryptographic operations due to blank nodes.

To conclude, this study shows that the DCGKA protocol offers a solution that is very efficient in the context of static groups, whereas DiSCreet, our distributed version of TreeKEM and Tainted TreeKEM protocol, offers a more scalable solution in case of highly dynamic groups. A more in-depth study as well as an experimental comparison, would be interesting to determine the exact criteria allowing one to decide on a protocol or the other.

6 Implementation

In this section we describe our implementation of DiSCreet[1] allowing MLS clients to communicate in a distributed manner.

Our implementation is written in C++ and is based on the open-source implementation of the MLS Protocol written by Cisco and named `MLS++`[2]. We then extend this implementation of the MLS Protocol by providing a distributed

[1]https://github.com/HiveNetCode/distributed-mls
[2]https://github.com/cisco/mlspp

Table 2: Comparison of the complexities for TreeKEM and Tainted TreeKEM using DiSCreet, and DCGKA, in the context of a dynamic group, whose members frequently issue *Add* and *Remove* operations. We consider the complexity of performing one operation, as well as concurrent operations, that would most likely be unintended.

Protocol	Complexity of cryptographic operations			Sent messages size for one process		Global number of exchanged messages
	Sender process[a]	Receiver process[b]	Subsequent operation[c]	Sender process[a]	Receiver process[b]	
Case 1: one dynamic operation (i.e. *Add* or *Remove*)						
TreeKEM with DiSCreet	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
Tainted TreeKEM with DiSCreet	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$
DCGKA	$\mathcal{O}(n)$	$\mathcal{O}(1)$	–	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n^2 \log n)$
Case 2: t concurrent dynamic operations						
TreeKEM with DiSCreet	$\mathcal{O}(t + t \log \frac{n}{t})$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2 + tn \log n)$
Tainted TreeKEM with DiSCreet	$\mathcal{O}(t + t \log \frac{n}{t})$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2 + tn \log n)$
DCGKA	$\mathcal{O}(n + t)$	$\mathcal{O}(t)$	–	$\mathcal{O}(n + t)$	$\mathcal{O}(t)$	$\mathcal{O}(tn^2 \log n)$

[a]Complexity for a member sending the commit in TreeKEM protocols, or submitting an operation in DCGKA.
[b]Complexity for a member that does not send the commit in TreeKEM protocols, or does not submit an operation in DCGKA.
[c]Complexity of submitting one *commit* after the n updates.

Delivery Service, without modifying the MLS implementation.

6.1 Architecture

We present the architecture of our implementation in Figure 8.

External components

As described in the MLS architecture specification [8], a Public Key Infrastructure (PKI) service is required to allow MLS clients to publish Key Packages. These packages contain users' identity keys as well as pre-keys which are necessary when inviting the user into an MLS group.

Additionally, in the context of our prototype the PKI also stores the addresses of each client. Therefore, Distributed MLS clients can use the PKI to acquire the knowledge of other members addresses and then establish a direct connection when needed.

In the case of this work, we do not focus on creating a distributed PKI, and it remains centralized. However, solutions exist for distributed PKIs such as [17] where a Distributed Hash Table (DHT) stores the public keys. Additionally, in a peer-to-peer environment, group members could find each other addresses in a distributed manner with mechanisms such as a DHT or *Rendez-vous points*.

Network communications

The network component of our implementation allows disseminating protocol messages in a distributed fashion. The component establishes TCP connections with necessary group members and thus avoids relaying any protocol message through a central server.

To acquire the knowledge of new group members addresses, the network component directly queries the PKI. Therefore, the other architecture components deal with addresses instead of members' identities.

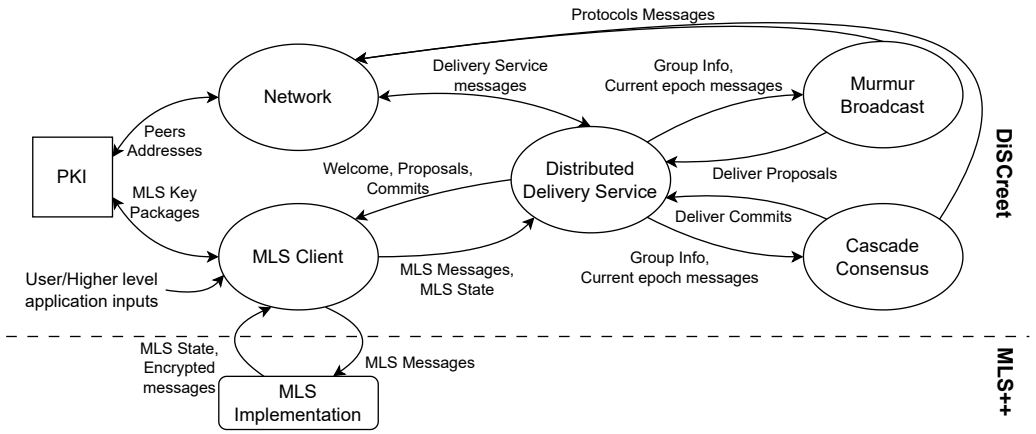


Fig. 8: Diagram depicting the components of our implementation and the interactions between these components.

MLS Client

The component called "MLS Client" encapsulates the open-source implementation of MLS. When an application client requests a group operation, the MLS Client component calls for the MLS implementation to generate proposals in the form of MLS messages. The Distributed Delivery Service further handles these MLS messages.

The Delivery Service sends to the MLS Client the proposals and commits that need to be applied at the right time and in the right order. The MLS Client notifies the Distributed Delivery Service when the group state evolves.

Additionally, the Delivery Service sends to the MLS Client a Welcome message in the case that a new client joins an MLS group.

Distributed Delivery Service

The Distributed Delivery Service whose functionality was described in Section 4.2 is the heart of the implementation.

It orchestrates the underlying communication mechanisms: the Murmur broadcast and the Cascade Consensus protocol. The Delivery Service forwards only valid messages received from the Network to these protocols, while delaying messages whose validity cannot be confirmed.

Finally, when the group state is updated, the Delivery Service notifies these sub-protocols. On one hand, the Murmur Broadcast aims at maintaining a gossip sample with enough members and eventually expands this sample by sending SUBSCRIBE messages to randomly picked members. On the other hand, the Cascade Consensus sub-protocols use the group size to compute the correct size of the *quorums* required to reach agreements and members' identity keys to verify signatures.

When the communication mechanisms determine the validity of a proposal or a commit message, these messages are delivered and directly forwarded to the MLS client.

6.2 Exchanging DiSCreet messages using MLS

In our implementation, to exchange messages we mostly reuse the structure and procedures provided by the MLS standard and its C++ implementation. The most interesting structures (from the ones described in Section 6 of the RFC [6]) are:

- *AuthenticatedContent*: designates the signed version of either a Proposal, a Commit or Application Data. It indicates which group member issued the message and ensures that it is indeed this user that sent the message.
- *MLSMessage*: designates the encrypted version of an *AuthenticatedContent* or eventually a Welcome message. It protects the messages exchanged inside the group and prevents anyone outside the group (i.e. without the knowledge of the current epoch's group key) to read the messages.

Then, the Distributed Delivery Service protocol encapsulates MURMUR, Cascade Consensus and Welcome messages. In the case of Proposals, the messages are already encoded as *MLSMessages*, so the MURMUR protocol does not need to apply any more encapsulation. However, we encapsulate the Cascade Consensus messages inside encrypted *MLSMessages*. This presents mainly two benefits: this protects the content of the messages, as well as tagging the messages with an epoch number, that allows the Delivery Service to effortlessly delay messages for future epochs. For signatures, required by the protocols of Cascade Consensus, we simply use MLS's *AuthenticatedContents*, which certifies the identity of its issuer.

This approach has the advantage of avoiding to introduce new complex data structures as well as modifying the MLS protocol itself. Additionally, we avoid introducing security issues, as we do not require the clients to keep new secrets or use new cryptographic protocols. However, for efficiency, one may benefit from a more in-depth integration of the distributed Delivery Service in the MLS Protocol. This is the case when using *Authenticated Contents* for signatures in Cascade Consensus as an *Authenticated Content* includes many irrelevant data such as the *group ID* or the current epoch. As these signatures reference a given commit, we already have the guarantee that such a signature cannot be used outside its context. The only exception is RETRACT messages in the Restrained Consensus whose content is only the string "RETRACT" and therefore it requires adding metadata to lock the message to a given context. Similarly, Cascade Consensus messages would benefit from a dedicated content type as in our implementation they are considered as *Application messages* in the MLS protocol and thus can be confused with the messages sent by the higher level applications using MLS.

7 Conclusion

In this paper we presented DiSCreet: a completely distributed solution for the design of the Delivery-Service component of TreeKEM by combining a Probabilistic Broadcast [18] method with the Cascade Consensus Protocol [1]. We formalized the role and the necessary properties of this Delivery-Service component, and we proposed a novel algorithm supporting its distributed counterpart. This design allows users to run the TreeKEM protocol or any similar Group Key Agreement protocol without requiring their communication to go through a central server. Our approach increases the security and the availability of group communication making it harder for an attacker to compromise the Delivery Service. Mounting attacks against the group requires compromising one third of the clients [1] instead of only one server in the standard centralized solution.

We showed that our approach is relevant in the context of dynamic groups by conducting a theoretical study comparing DiSCreet with the DCGKA protocol [31]. We also presented an implementation of DiSCreet based on an open-source implementation of MLS.

We plan to determine which attacks can be performed to target a Delivery Service in the general case (either centralized or distributed). Such a study will lay the groundwork for a formal analysis of the security of our solution. Additionally, we plan to conduct a performance evaluation of our solution, by comparing it with a centralized version of the Delivery Service, as well as with other solutions for distributed Group-Key Agreement. Then, we will further extend our solution in order to explicitly support offline group members. Currently, the TreeKEM protocol supports offline members using a central server that stores messages for these users. We plan to design completely distributed alternative solutions for offline communication by relying on State-Machine-Replication [7] mechanisms such as State Transfer. This should allow previously offline members to get up to date with the group by directly synchronizing with some other members. In order to support a completely distributed group-key-management component for our real-time peer-to-peer collaborative editor MUTE [24], we plan to integrate the TreeKEM protocol and our distributed Delivery Service. We further plan to combine this distributed group-key-management mechanism with a distributed access control mechanism such as ACCURE [27].

Acknowledgements

This work is supported by the "Alvearium" Inria and hive partnership.

Conflicts of interest

Not applicable.

References

[1] Albouy T, Frey D, Gestin M, et al (2023) Context adaptive cooperation. URL https://arxiv.org/abs/2311.08776, 2311.08776

[2] Alwen J, Coretti S, Dodis Y, et al (2021) Modular design of secure group messaging protocols and the security of mls. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, CCS '21, p 1463–1483, https://doi.org/10.1145/3460120.3484820

[3] Alwen J, Auerbach B, Noval MC, et al (2022) Cocoa: Concurrent continuous group key agreement. In: Advances in Cryptology – EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 – June 3, 2022, Proceedings, Part II. Springer-Verlag, Berlin, Heidelberg, p 815–844, https://doi.org/10.1007/978-3-031-07085-3_28

[4] Ateniese G, Steiner M, Tsudik G (1998) Authenticated group key agreement and friends. In: Proceedings of the 5th ACM Conference on Computer and Communications Security, CCS '98, p 17–26, https://doi.org/10.1145/288090.288097

[5] Balbás D, Collins D, Gajland P (2022) Analysis and improvements of the sender keys protocol for group messaging. XVII Reunión española sobre criptología y seguridad de la información RECSI 2022 265:25. URL https://arxiv.org/abs/2301.07045

[6] Barnes R, Beurdouche B, Robert R, et al (2023) The Messaging Layer Security (MLS) Protocol. RFC 9420, https://doi.org/10.17487/RFC9420

[7] Bessani A, Sousa J, Alchieri EE (2014) State machine replication for the masses with bft-smart. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, pp 355–362, https://doi.org/10.1109/DSN.2014.43

[8] Beurdouche B, Rescorla E, Omara E, et al (2022) The Messaging Layer Security (MLS) Architecture. Internet-Draft draft-ietf-mls-architecture-10, Internet Engineering Task Force, URL https://datatracker.ietf.org/doc/draft-ietf-mls-architecture/10/, work in Progress

[9] Bhargavan K, Barnes R, Rescorla E (2018) TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). Research report, Inria Paris, URL https://hal.inria.fr/hal-02425247

[10] Bienstock A, Dodis Y, Rösler P (2020) On the price of concurrency in group ratcheting protocols. In: Pass R, Pietrzak K (eds) Theory of Cryptography. Springer International Publishing, Cham, pp 198–228, URL https://eprint.iacr.org/2020/1171

[11] Bracha G (1987) Asynchronous byzantine agreement protocols. Information and Computation 75(2):130–143. https://doi.org/https://doi.org/10.1016/0890-5401(87)90054-X

[12] Brzuska C, Cornelissen E, Kohbrok K (2022) Security analysis of the mls key derivation. In: 2022 IEEE Symposium on Security and Privacy (SP), pp 2535–2553, https://doi.org/10.1109/SP46214.2022.9833678

[13] Burmester M, Desmedt Y (1995) A secure and efficient conference key distribution system. In: De Santis A (ed) Advances in Cryptology — EUROCRYPT'94, URL https://www.cs.fsu.edu/~langley/Eurocrypt/euro-pre.pdf

[14] Castro M, Liskov B, et al (1999) Practical byzantine fault tolerance. In: 3rd Symposium on Operating Systems Design and Implementation (OSDI 99). USENIX Association, New Orleans, LA, URL https://www.usenix.org/conference/osdi-99/practical-byzantine-fault-tolerance

[15] Cohn-Gordon K, Cremers C, Garratt L, et al (2018) On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, CCS '18, p 1802–1819, https://doi.org/10.1145/3243734.3243747

[16] Fischer MJ, Lynch NA, Paterson MS (1985) Impossibility of distributed consensus with one faulty process. J ACM 32(2):374–382. https://doi.org/10.1145/3149.214121

[17] Fromknecht C, Velicanu D, Yakoubov S (2014) A decentralized public key infrastructure with identity retention. IACR Cryptol ePrint Arch 2014:803. URL https://eprint.iacr.org/2014/803

[18] Guerraoui R, Kuznetsov P, Monti M, et al (2019) Scalable Byzantine Reliable Broadcast (Extended Version). In: 33rd International Symposium on Distributed Computing (DISC 2019), URL https://arxiv.org/abs/1908.01738

[19] Kim Y, Perrig A, Tsudik G (2000) Simple and fault-tolerant key agreement for dynamic collaborative groups. In: Proceedings of the 7th ACM Conference on Computer and Communications Security, CCS '00, p 235–244, https://doi.org/10.1145/352600.352638

[20] Klein K, Pascual-Perez G, Walter M, et al (2021) Keep the dirt: Tainted treekem, adaptively and actively secure continuous group key agreement. In: 2021 IEEE Symposium on Security and Privacy (SP), pp 268–284, https://doi.org/10.1109/SP40001.2021.00035

[21] Lamport L (2001) Paxos made simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) pp 51–58. URL https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[22] Malki D, Reiter M (1996) A high-throughput secure reliable multicast protocol. In: Proceedings 9th IEEE Computer Security Foundations Workshop, pp 9–17, https://doi.org/10.1109/CSFW.1996.503686

[23] Moxie M, Trevor P (2016) Signal - specifications - the x3dh key agreement protocol. URL https://signal.org/docs/specifications/x3dh/

[24] Nicolas M, Elvinger V, Oster G, et al (2017) MUTE: a peer-to-peer web-based real-time collaborative editor. In: ECSCW 2017 - 15th European Conference on Computer-Supported Cooperative Work, Proceedings of 15th European Conference on Computer-Supported Cooperative Work - Panels, Posters and Demos, vol 1. EUSSET, Sheffield, United Kingdom, pp 1–4, https://doi.org/10.18420/ecscw2017_p5

[25] Paillat L, Ignat CL, Frey D, et al (2023) Design of an efficient distributed delivery service for group key agreement protocols. In: International Symposium on Foundations and Practice of Security, Springer

[26] Perrin T, Marlinspike M (2016) The double ratchet algorithm. Signal - Specifications, URL https://signal.org/docs/specifications/doubleratchet/

[27] Rault PA, Ignat CL, Perrin O (2023) Access control based on CRDTs for Collaborative Distributed Applications. In: The International Symposium on Intelligent and Trustworthy Computing, Communications, and Networking (ITCCN-2023), Proceedings of the 22nd IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom-2023), Exeter, UK, URL https://inria.hal.science/hal-04224855

[28] Steiner M, Tsudik G, Waidner M (2000) Key agreement in dynamic peer groups. IEEE Transactions on Parallel and Distributed Systems 11(8):769–780. https://doi.org/10.1109/71.877936

[29] Wallez T, Protzenko J, Beurdouche B, et al (2023) Treesync: Authenticated group

management for messaging layer security. URL https://www.usenix.org/system/files/sec23fall-prepub-372-wallez.pdf, prepublication 32nd Usenix Security Symposium

[30] Weidner M (2019) Group messaging for secure asynchronous collaboration. Master's thesis URL https://mattweidner.com/assets/pdf/acs-dissertation.pdf

[31] Weidner M, Kleppmann M, Hugenroth D, et al (2021) Key agreement for decentralized secure group messaging with strong security guarantees. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, CCS '21, p 2024–2045, https://doi.org/10.1145/3460120.3484542