Replacing Ownly Group Encryption with OpenMLS

Xuanzhe Han
UCLA
USA
xuanzhehan2002@g.ucla.edu

Abstract

Ownly is a decentralized collaborative workspace built on the Named Data Networking (NDN) stack. Its original workspace encryption derives a static AES-GCM key from a pre-shared key (PSK) and a dynamic shared key (DSK), a design that fits the existing application but does not provide clean semantics for membership changes, epoch transitions, forward secrecy, or post-compromise recovery. This report presents the design, implementation, and evaluation of an MLS-based replacement for that scheme using OpenMLS. Rather than replacing Ownly's entire data plane with native MLS ciphertexts, the final system uses a Rust/WebAssembly OpenMLS module to manage group state, process KeyPackage, Commit, and Welcome messages, and export one shared workspace key per epoch, while preserving the existing TypeScript frontend, Go WebAssembly NDN backend, and SVS/repository-based publication pipeline. The implementation adds MLS control-message transport over the existing synchronization path, browser-side persistence and restoration of MLS state, and application-level mechanisms for multi-device owner sequencing, reset and rejoin, and encrypted-state recovery across key rotations. Evaluation on the completed local deployment shows that the system supports member addition and removal, multi-device participation, restart recovery, and group-wide MLS reset without requiring a redesign of the surrounding NDN architecture. The main result is that MLS can be integrated into Ownly as a practical group key-management layer, but doing so requires explicit application-level policies for sequencing, persistence, and recovery in addition to the protocol itself.

1 Introduction

Ownly is a decentralized collaborative workspace built on the Named Data Networking (NDN) stack. Its frontend runs in TypeScript, while NDN networking, publication, and storage logic are exposed through a Go WebAssembly backend. Real-time dissemination uses State Vector Sync (SVS), and larger objects are stored and retrieved through an NDN repository service. Before this project, Ownly protected workspace traffic with a symmetric AES-GCM key derived from two inputs: a pre-shared key (PSK) and a dynamically shared key (DSK). The PSK is a secret distributed through the workspace join flow, in practice through the invitation link used by members to join the workspace. The DSK is a workspace secret generated when the workspace is created and later obtained by joining members from currently online participants through a DSK exchange protocol. These two values are combined to derive the single AES-GCM key used to encrypt workspace publications.

This design is simple and fits naturally into Ownly's existing architecture, but it has clear security limitations. The derived workspace key is effectively static for the lifetime of the workspace. Membership changes do not automatically rotate the key, so joins and removals do not create a clean cryptographic boundary. A new member who obtains the current workspace key can decrypt older content that was encrypted under that same key, and a removed member who still knows the key can continue decrypting future traffic until someone manually rekeys the workspace. More generally, compromise of that long-lived key compromises both past and future workspace traffic. In short, the legacy design provided no forward secrecy, no post-compromise recovery, and no rekeying semantics tied to group events.

Messaging Layer Security (MLS), standardized by the IETF in RFC 9420 [1], was designed to address exactly these problems for end-to-end encrypted groups. At a high level, MLS organizes the group into epochs and advances the group state whenever members are added, removed, or a member invokes an update to advance the group state for any reason. Each epoch yields fresh shared secret material, and the protocol's TreeKEM ratchet lets members derive those secrets efficiently while providing forward secrecy and post-compromise security. Although MLS can provide encryption and decryption directly, adopting the full MLS record layer would require replacing much more of Ownly's existing publication pipeline than this project needs. Ownly already has a working mechanism for encrypting and distributing workspace objects over SVS and repository-backed blobs; the missing piece is not a new data plane, but a principled group key-management layer. Full MLS also derives multiple secrets per epoch and maintains sender-specific key and nonce schedules for per-message encryption. That mechanism is useful for a strict per-message security guarantee, but Ownly treats every user as equal and such security guarantee is not necessary.

This project therefore integrates OpenMLS, an open-source implementation of MLS, into Ownly as the mechanism for group state evolution and key derivation, while preserving the surrounding NDN application architecture. Rather than sending application messages as native MLS ciphertexts, the system uses MLS to manage group membership, process KeyPackage, Commit, and Welcome messages, and export a fresh 32-byte shared secret for each epoch. That exported secret becomes the active workspace encryption key used by Ownly's existing AES-GCM path. This approach preserves the TypeScript frontend, the Go WebAssembly NDN backend, and the SVS/repository-based publication flow, while replacing the static PSK+DSK scheme with membership-driven key rotation.

The final system therefore shifts Ownly's workspace encryption from an epochless shared-secret model to a protocol-managed group state model. In doing so, it introduces not only MLS-based membership semantics, but also a new set of engineering concerns around persistence, restart recovery, multi-device coordination, and key retention across snapshots. The rest of this report presents the design and implementation of this MLS integration, evaluates its behavior under realistic workspace workflows, and discusses the systems-level tradeoffs required to embed MLS into an existing decentralized collaborative application.

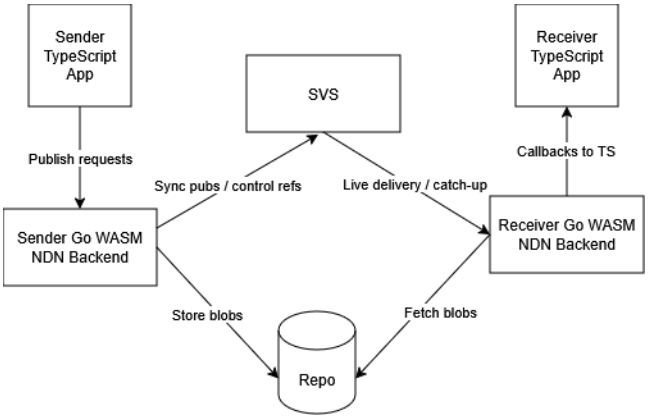

Figure 1: Original Ownly data path. TypeScript issues publish requests to the Go WASM NDN backend, SVS distributes live synchronization and control references, and the repository stores blobs for later retrieval.

2 System Background

2.1 Ownly Background

Ownly is a browser-based collaborative workspace built on the Named Data Networking (NDN) stack. The frontend is implemented in TypeScript, while NDN functionality is exposed through a Go WebAssembly backend. The TypeScript application invokes this backend to join workspaces, publish synchronized updates, fetch named objects, and manage subscriptions. Real-time dissemination uses State Vector Sync (SVS), which carries small synchronization and control references, while larger objects are stored as named blobs in an NDN repository service and fetched by peers as needed.

Figure 1 summarizes the original Ownly data path. On the sending side, TypeScript issues publication requests to the Go WebAssembly backend. The backend publishes either encrypted application data or small control references through SVS. When the payload is too large or should be retrieved later, the raw bytes are stored as repository-backed blobs and referenced by name. On the receiving side, a peer's Go backend receives SVS publications, resolves any referenced blobs from the repository, and then invokes TypeScript callbacks that update local application state.

Before this project, workspace encryption followed a legacy PSK+DSK design. The PSK is a pre-shared secret distributed through the workspace join flow, for example in the invitation link. The DSK is a dynamically shared secret generated when the workspace is created and later obtained by joining members from currently online participants through a DSK exchange protocol. These two values are combined to derive a single AES-GCM workspace key, and that key is used to encrypt synchronized workspace publications.

This design fit naturally into Ownly's existing architecture, but it made the workspace key effectively static: joins, removals, and other membership changes did not automatically create a new cryptographic epoch or rotate the key. At the same time, the surrounding mechanisms for content snapshots, symmetric encryption, and NDN-based data transport were already in place. As a result, the project did not replace the surrounding publication stack; instead, it targeted only the mechanism that generates and updates the workspace encryption key.

2.2 MLS Background

Messaging Layer Security (MLS) is the IETF-standardized protocol for secure group messaging [1]. MLS is designed to provide forward secrecy and post-compromise security for dynamic groups whose membership changes over time. An existing open-source implementation of MLS is OpenMLS, a Rust library that implements the core protocol state machine and cryptographic operations [2]. In this project, OpenMLS serves as the underlying MLS engine integrated into Ownly through WebAssembly.

At a high level, MLS organizes a group into *epochs*. Each epoch represents a consistent version of the group state and yields fresh shared secret material for the current members. When a member is added, removed, or updated, the group advances to a new epoch. This is the key difference from Ownly's legacy PSK+DSK design: in MLS, membership changes are tied directly to cryptographic state transitions instead of relying on manual rekeying outside the protocol.

Compared to a naive rekeying scheme that updates members one by one in $O(N)$ time, MLS can update the group more efficiently through TreeKEM [1]. Group members occupy the leaves of a logical binary tree, and each update refreshes only the secrets along one path from a leaf to the root. The updated path secrets are then distributed to the relevant sibling subtrees using encrypted update material. As a result, MLS does not need to rekey every member individually after each change; instead, the work grows logarithmically with group size. For an MLS key update in a balanced tree, the cost is therefore typically described as $O(\log N)$ in a group of N members.

MLS distinguishes several protocol messages that drive group evolution. A joining member contributes a *KeyPackage*, which advertises the cryptographic material needed to add that member to the group. An existing member designated as the committer produces a *Commit*, which advances the group to a new epoch and incorporates additions, removals, or self-updates. Newly added members receive a corresponding *Welcome* message, which gives them the information required to initialize the same group state as the existing members. Current members apply the Commit, while invitees join by processing the Welcome.

In the full MLS protocol, each epoch derives multiple secrets for handshake traffic, application-data protection, and sender-specific key and nonce schedules. This provides a stronger native message-protection model than Ownly needs. In Ownly, the important requirement is not native MLS framing for every payload, but a reliable way to evolve the shared workspace key whenever the group changes. OpenMLS exposes exporter-based key derivation from the current group state, which makes it possible to use MLS as the group key-management layer while reusing Ownly's existing AES-GCM content path.

3 Project Design and Implementation

3.1 High-Level Approach

The final design preserves Ownly's existing publication and synchronization pipeline and replaces only the mechanism that generates and rotates the workspace encryption key. The TypeScript frontend, Go WebAssembly NDN backend, SVS-based synchronization path, repository-backed blob transport, and AES-GCM content encryption all remain in place. MLS is used as the group state machine: when the group changes, the local MLS state advances to a new epoch and exports a fresh 32-byte secret. That exported secret is then installed as the active workspace encryption key through the existing Go backend interface.

This separation keeps the integration narrow. Membership semantics, epoch transitions, and rekeying are delegated to MLS, while data transport and document synchronization continue to use the mechanisms Ownly already had. The legacy PSK+DSK path is still used for initial workspace bootstrap and as a fallback during MLS recovery, but steady-state group key evolution is driven by MLS epochs rather than by a static shared secret.

3.2 Rust/OpenMLS WebAssembly Integration

To run MLS inside the browser, the project introduces a Rust module that is compiled to WebAssembly and loaded by the TypeScript frontend. The module wraps OpenMLS and exposes only the operations needed by Ownly:

- create a group,
- generate a KeyPackage,
- join from a Welcome message,
- add and remove members,
- apply or merge Commit messages,
- export a 32-byte epoch secret,
- export and import OpenMLS storage,
- reload a persisted group by group identifier.

The Go backend exports the current workspace certificate to the frontend, and the TypeScript wrapper constructs the MLS client from two pieces of identity material: a stable application-level device identity string of the form `NDN name/device ID`, where the device ID is a random value persisted for that device, and the final workspace certificate currently issued by Ownly's trust layer. The Rust module serializes both values into the MLS BasicCredential payload, so the current implementation can continue to use the account/device identity for routing and device management while also carrying workspace-issued certificate material inside MLS credentials.

Figure 2 shows the resulting architecture. The Rust/WebAssembly module maintains MLS group state and key evolution, TypeScript orchestrates policy and persistence, and the Go backend continues to publish references and fetch blobs over NDN.

3.3 MLS Control Plane over Existing Transport

The implementation does not introduce a separate MLS transport. Instead, it extends Ownly's existing SVS/blob publication path with three MLS control reference types:

- KeyPackage reference,
- Welcome reference,
- Commit reference.

The raw MLS objects themselves are stored as blobs through the existing repository-backed mechanism. SVS carries only the references: the invitee identity, the blob name, and a session identifier. The session identifier is encoded as `groupId:epoch`, which lets the application discard stale commits, queue out-of-order commits, and distinguish commits from an older group instance after reset or rejoin. On the receiving side, the Go backend decodes MLS references from live publications and compressed snapshots, passes them to TypeScript in batches, and the TypeScript layer deduplicates them using the publisher, boot time, and sequence number metadata already attached to SVS publications.

This reuse of the existing transport path is a central design choice. It avoids a second networking stack, keeps MLS message distribution aligned with the rest of Ownly's synchronization model, avoids separate authentication logic for MLS-specific payloads, and allows the application to treat KeyPackage, Welcome, and Commit objects as just another class of named blobs plus small synchronization references.

3.4 Membership and Session Management

Membership is managed through an explicit control-plane flow coordinated by the owner. Since each physical device is treated as an independent MLS member, one owner device (the device used to create the workspace) is designated as master owner device. The user can manually change this designation. The master owner device lazily bootstraps the owner MLS group the first time it needs to perform an access-management action. Non-owner devices, and owner devices that are not currently the designated master, request MLS participation by publishing a KeyPackage blob and an SVS reference to it.

The add-member path works as follows:

(1) A prospective member publishes a KeyPackage blob and a KeyPackage reference.
(2) The master owner device receives the reference, fetches the blob, and extracts the embedded MLS identity.
(3) The owner checks that the account portion of that identity is already authorized in the existing invitation map stored in shared Yjs state.
(4) The owner adds the member to the MLS group, merges the pending Commit locally, exports a fresh 32-byte epoch secret, and installs it as the new workspace key.
(5) The owner publishes a Commit reference to the group and a Welcome reference targeted at the invitee.
(6) The invitee fetches the Welcome blob, joins the MLS group, and installs the same epoch key.
(7) Existing members fetch the Commit blob, apply it, and install the updated epoch key.

Figure 3 shows this workflow. In the report figures and code, the committer is the *master owner device*, not just an arbitrary owner device.

The remove-member path is similar but produces only a Commit. The master owner device resolves the MLS leaves belonging to the target account, removes them from the MLS group, merges the pending Commit locally, rotates the workspace key, and publishes a Commit reference targeted at the removed account. When devices

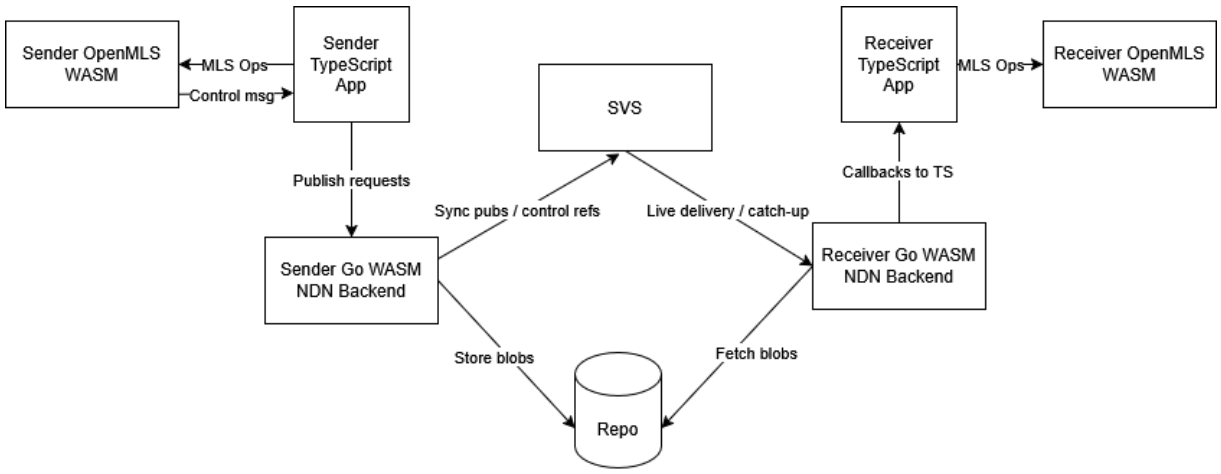

Figure 2: Updated Ownly data path with MLS integration. TypeScript drives the Rust/OpenMLS WebAssembly module for group-state operations, while the Go WASM NDN backend continues to carry SVS control references and repository-backed blob transport.

under that account receive the targeted removal Commit, they revoke their local MLS state, mark the workspace as revoked, and are forced to leave the workspace UI until they rejoin with a fresh invitation.

This design also accounts for users having multiple devices. Each device is treated as an independent MLS member, with its own MLS identity of the form `NDN name/device ID`. As mentioned before, the device ID is a random value assigned to the device on workspace initialization, guaranteeing naming uniqueness. For owner devices specifically, shared Yjs state stores a registry of owner devices and the current master owner device identifier. All commit-generating operations—inviting members, removing members, and resetting MLS state—are restricted to the current master owner device. This is necessary because MLS assumes the application will ensure that only one device performs commit merging at a time; otherwise, conflicts may lead to inconsistent MLS state machines and members holding different secrets. The global owner device registry avoids conflicting local merges by multiple owner devices while still allowing master-role transfer and per-device labels for administration.

3.5 Persistence, Recovery, and Rejoin

Persistence is required because the application runs in the browser and the MLS state machine would otherwise disappear on refresh or restart. The implementation persists two MLS artifacts in the browser-side state database:

- an exported OpenMLS storage snapshot, and
- the MLS group identifier.

On startup, the application checks whether both artifacts exist. If they do, the TypeScript layer reconstructs the OpenMLS client, imports the stored key-value snapshot, reloads the MLS group by group identifier, reconstructs the corresponding signing key from persisted OpenMLS storage, reinstalls the current workspace key, and then applies any queued Commit references that were received while the client was offline. The design explicitly rejects partially present state: if only one of the two persistence artifacts exists, the state is treated as incomplete and recovery is required.

The implementation also supports several recovery cases beyond a clean restart. Local MLS state can be reset by the master owner device and propagated to the rest of the group through a dedicated MLS reset sentinel. Non-owner devices clear their local MLS state on reset and return temporarily to the legacy PSK+DSK key path until they request MLS participation again. A directed reset request path also allows non-owner devices to ask the current master owner device to perform a group-wide MLS reset when consistency is lost.

3.6 Key Rotation and Snapshot Availability

One of the main systems challenges is that Ownly synchronizes data through encrypted Yjs updates and snapshots that may remain in transit or be replayed after the group key has already rotated. To keep those older updates readable during a short recovery window, the workspace metadata stores a small rolling window of recent MLS session keys, each indexed by its `groupId:epoch` session identifier. On startup, those recent keys are reinstalled into the Go backend before the workspace begins processing publications.

This rolling key window is paired with explicit encrypted-state republishing. After the master owner device advances the MLS session, it republishes the current encrypted workspace state so that idle or late members can fetch content under the newest session key. The same republish mechanism is exposed to the owner and to members through a request called "SOS" as a recovery mechanism for cases where a user gets stuck on older snapshots. In other words, MLS determines when the workspace key changes, but the application is still responsible for making sure encrypted state under that new key becomes available to peers.

3.7 Implemented Features

The completed implementation now includes:

- Rust/OpenMLS integration through WebAssembly with the P-256 MLS ciphersuite,

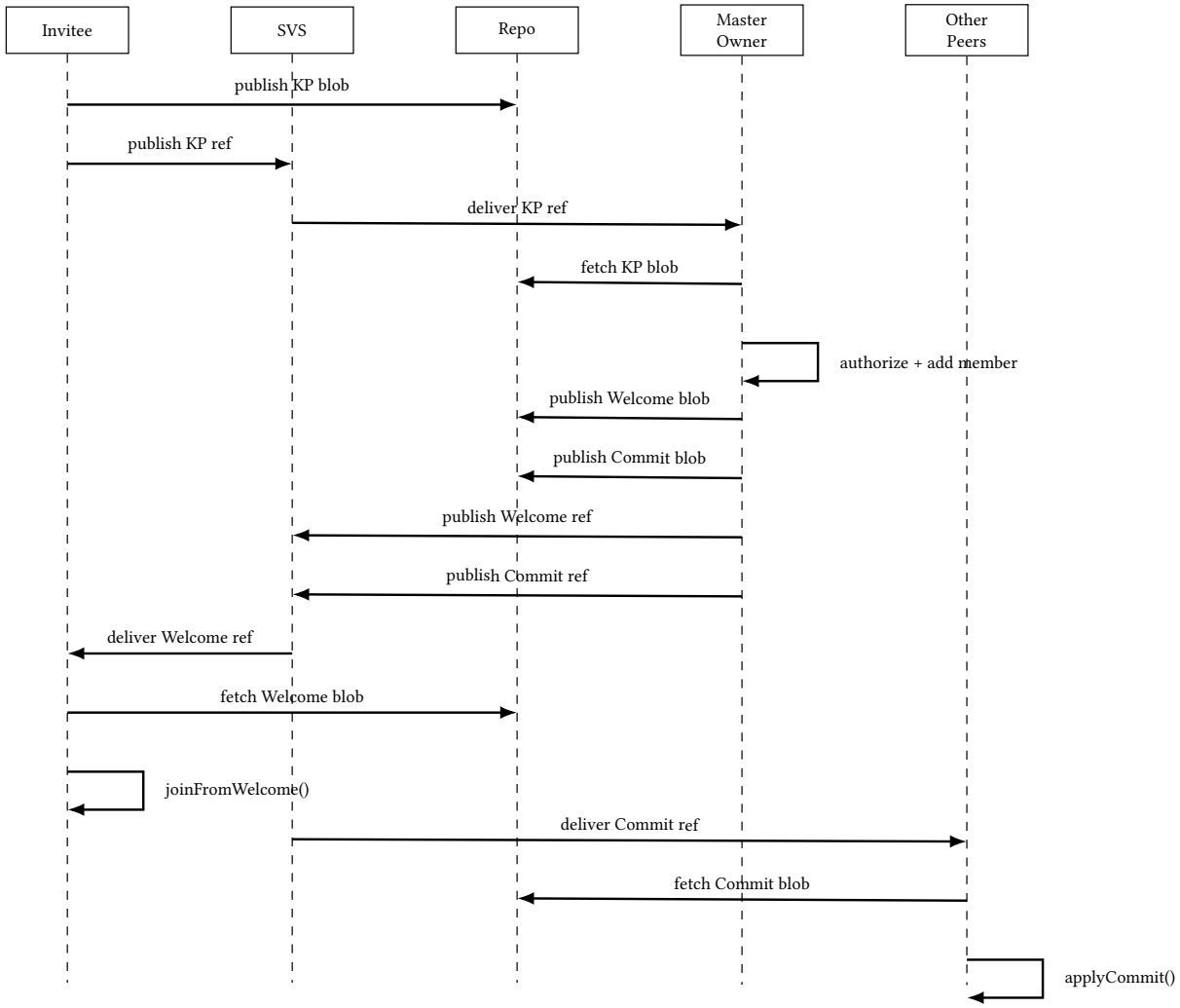

Figure 3: Membership update flow over the existing Ownly transport. The invitee publishes a KeyPackage reference, the master owner device generates Welcome and Commit messages, the invitee joins from the Welcome, and other peers fetch and apply the Commit to advance group state.

- transport of KeyPackage, Welcome, and Commit references over the existing SVS/blob pipeline,
- export of one 32-byte workspace key per MLS epoch into the existing AES-GCM encryption path,
- owner-controlled member addition and removal,
- multi-device owner coordination through a single master owner device,
- persistence and restoration of MLS storage across browser restarts,
- rolling retention of recent session keys for delayed snapshot decryption,
- encrypted-state republishing and directed SOS refresh for recovery, and
- group-wide MLS reset and revoked-member ejection handling.

Taken together, these mechanisms move Ownly from a static PSK+DSK-derived workspace key to a design in which membership changes are tied to explicit MLS epoch transitions, while still preserving the application's original NDN publication pipeline and encrypted content format.

Appendix A includes selected code excerpts for the MLS control-message format, account/device identity handling, and directed MLS reset path.

4 Evaluation

4.1 Methodology

The evaluation focuses on the integrated system rather than on isolated cryptographic primitive costs. MLS itself is implemented by OpenMLS; the contribution of this project is the surrounding browser, NDN, synchronization, persistence, and recovery logic

required to make MLS usable inside Ownly. Accordingly, the evaluation is divided into two parts:

- **functional evaluation**, which checks whether the implemented workflows behave correctly under realistic workspace scenarios
- **timing evaluation**, which measures the latency of the main MLS-driven workflows as observed by the running application.

Functional tests were performed on a deployed system. All timing tests were performed on the local system. Because the available test environment was limited to a small number of browser instances and three email-backed identities, repeated timing samples were collected by exercising the same workspace through repeated add/remove/reinvite and reset/rejoin cycles rather than by creating a completely fresh deployment for every run. The reported values therefore reflect realistic repeated use of the same local deployment.

To support the timing study, the application was instrumented with browser-side timers around the main MLS workflows. These timers record end-to-end durations observed by the application, including local OpenMLS processing, serialization, blob publication and retrieval, workspace-key installation, and the surrounding TypeScript orchestration. They therefore measure the cost of the integrated Ownly workflow rather than the cost of OpenMLS alone. The timing table reports the average, median, and maximum observed latency over 10 runs.

4.2 Functional Evaluation

The first part of the evaluation checks whether the final system behaves as intended under the workflows that motivated the MLS integration. Table 1 summarizes the tested scenarios and the observed results.

Across these tests, no functional failures were observed in the final deployed workflow. In particular, the combination of MLS-driven key rotation, browser-side persistence, explicit owner-device sequencing, and recovery helpers was sufficient to support membership changes and repeated rejoin cycles.

4.3 Timing Evaluation

The second part of the evaluation measures the latency of the main MLS-driven workflows. To collect timing data, I added a browser-side timing helper to the application and exposed it through the JavaScript console. The collected timing data is shown in Table 2. Because the helper is browser-local, the same workflow can be measured separately from the owner, invitee, or existing member point of view depending on which browser instance has the collector enabled. For this report, I ran the collector on the owner browser and one member browser.

These results show that the internal MLS operations were not the dominant source of latency in the tested deployment. On the owner side, adding a member averaged 21.4 ms, and removing a member averaged 130.6 ms. On the member side, joining from a received Welcome averaged only 9.9 ms, and applying a Commit averaged 13.3 ms. In other words, once the required MLS objects were available locally, group-state evolution and workspace-key rotation were lightweight.

The larger costs came from the surrounding system. The member-side `workspace.create` path averaged 5376.5 ms, and the `workspace.find_dsk` step alone averaged 3099.5 ms. This indicates that in the current local deployment, startup and bootstrap overhead is dominated by existing Ownly and NDN coordination steps rather than by MLS itself. Similarly, both owner bootstrap and KeyPackage publication showed large outliers, suggesting occasional cold-start or publication-path variance even when the median case remained low.

4.4 Discussion

The evaluation shows that MLS can be integrated into Ownly without replacing the surrounding NDN publication path. Functionally, the completed system supports member addition, removal, multi-device participation, restart recovery, reset, and rejoin workflows under the deployed browser-based model. Timing measurements further show that the local MLS operations themselves are relatively inexpensive once the required objects are available. The larger costs instead arise from the surrounding system: DSK acquisition, startup, publication delay variance, and encrypted-state recovery.

Overall, the results match the main lesson from the implementation: replacing the legacy PSK+DSK scheme was not just a cryptographic change. Most of the work came from making MLS fit into a decentralized collaborative application. Sequencing, persistence, state republication, and recovery policy all affected the user-visible behavior, and in the tested deployment they dominated the cost of the pure MLS state transitions.

5 Implementation Challenges

Several hard problems appeared only after the prototype started working. These were not obvious at the beginning of the project.

5.1 Bootstrap Ordering and Join Material Delivery

A major issue is bootstrap ordering. If the workspace transport switches to the MLS-derived key too early, a new invitee may not yet have the information needed to join the MLS group. In particular, the user may need to fetch onboarding material before the MLS state is available locally. This creates a bootstrapping dependency: MLS is supposed to provide the new key, but the system still needs an earlier path to deliver the information required to join MLS in the first place.

This means that migration cannot simply be “flip the key source”. The onboarding path must be sequenced carefully. In practice, the design uses the legacy path to start onboarding, and then switches to OpenMLS state once MLS is initialized on the participating devices.

5.2 Snapshot Handling Across Key Rotations

The most important issue is snapshots. In the current Ownly design, synchronized data and snapshots are encrypted. Once the workspace key starts rotating with MLS epochs, snapshots become much harder to reason about. An offline member may later fetch a snapshot containing entries encrypted under an old session key, while only knowing the current one. This can make catch-up fail even if the MLS state machine itself restores successfully.

Table 1: Functional evaluation of the final MLS-integrated Ownly workflow.

Scenario	Result
Add member	Working; Welcome and Commit were processed successfully.
Remove member	Working; removed members lost access to new encrypted state.
Multi-device member join	Working; distinct devices joined under the same account authorization.
Multi-device owner coordination	Working; only the master owner device generated commits.
Browser restart recovery	Working; persisted MLS state was restored after restart.
MLS state reset	Working; reset and rejoin completed successfully.
Reinvite after removal	Working; a fresh invitation restored access correctly.
Snapshot/key-rotation recovery	Working; republish plus rolling keys preserved usability after rekey.

Table 2: Timing evaluation of the main MLS-driven workflows.

Role	Workflow	Avg. (ms)	Median (ms)	Max (ms)
Owner	Owner MLS bootstrap	308.1	8	2103
Owner	Add member	21.4	18	37
Owner	Remove member	130.6	109	244
Owner	Group-wide MLS reset	91.3	88	108
Owner	Republish encrypted state	56.6	49	160
Owner	Workspace startup	283.7	284	323
Owner	Wait for user key	7.7	6	15
Owner	Restore persisted MLS state	17.0	17	17
Member	Publish KeyPackage reference	550.4	26	2131
Member	Join request to Welcome	252.1	253	287
Member	Join from Welcome	9.9	9	12
Member	Apply Commit	13.3	13	16
Member	Reset local MLS state	61.7	29	214
Member	Workspace startup	5376.5	4322	6431
Member	Find DSK	3099.5	2088	4111
Member	Wait for user key	3.5	3	4

The solution is to keep a short 5-key rolling window for delayed/out-of-order snapshot entries. On owner-driven key rotation, the owner publishes a fresh group snapshot encrypted with the new key. This scheme gives an offline member a recent snapshot to fetch when they come back online. This is also one reason why a force-snapshot request was added. It allows a user to broadcast a snapshot request ping to online users. Online users respond with a pong, and the requester can choose a responder and ask them directly to publish a new snapshot. This mechanism is separate from the MLS implementation, and was originally designed to solve a prior issue where a user fetches a stale snapshot and gets stuck. It also serves as a recovery mechanism for the MLS integration. If a member fetches an old snapshot that was encrypted using a session key outside the rolling key window, this mechanism lets them recover by forcing the generation of a new snapshot they can fetch.

5.3 Logical Ownership and Multi-Device Commit Ordering

One design issue that became clear during the project is that identity is logical but MLS membership is physical. A user can own multiple devices, and each device is treated as an independent MLS member since each needs its own local MLS state machine. However, a user is a logical entity, and all devices the user owns are managed under the same NDN name. This complicates the story since MLS commits form a state machine and must be applied in order. The MLS protocol assumes this ordering is guaranteed by the application. It makes sense to have the owner, who is responsible for inviting new members, generate commits, but a logical owner may operate multiple devices, and those devices may all be valid participants in the same workspace. If all of these devices are allowed to generate and merge commits, ordering is hard to enforce.

This points to the need for coordination among owner devices to guarantee ordering. If membership-changing MLS commits are generated and merged by a single device, Ownly follows the MLS requirement of having one state machine merge commits at a time, and transport ordering becomes much simpler. This is why the implementation designates a master owner device to perform invitation and generate commits. A mechanism like a global owner device registry is necessary to delegate this role and facilitate coordination.

6 Future Work

The final system is functionally complete enough to support the intended MLS workflows, but several follow-up directions remain.

First, the current evaluation is intentionally small-scale. It demonstrates that the deployed system works across member addition, removal, multi-device participation, restart recovery, and MLS reset, but it does so in a local environment with a limited number of accounts and browser instances. A natural next step is a broader evaluation with more users, more devices per user, longer-running sessions, and more repeated membership churn. That would make it possible to study how the current design behaves under heavier sustained use rather than only under the moderate workflows exercised in this report.

Second, the current snapshot and key-retention policy should be refined further. The final design already keeps a small rolling window of recent session keys and republishes encrypted state after MLS-driven key rotation, which was sufficient for the tested deployment. However, this policy is still heuristic. A stronger follow-up study would examine how many recent session keys need to be retained in practice, when those keys can be discarded safely, and how often encrypted state should be republished to balance recovery convenience against storage and publication overhead.

Third, the owner-device registry should likely be revisited. In the current design, metadata about owner devices and the designated master owner device is stored in shared Yjs state. This is convenient for synchronization, but it may not be the most principled place to keep authority-related device metadata. A possible alternative is to infer the set of owner devices directly from MLS participation state instead of maintaining a separate shared registry. Under that model, an owner device that joins an existing workspace through the invitation path would implicitly be treated as a non-master owner device unless authority is explicitly transferred to it.

This change would also make master-device transfer more explicit. Instead of treating master ownership as a replicated shared-state flag that can move independently of liveness, transfer could require both the current master device and the new target device to be online at the same time. That would better align the authority transfer with an active handoff between two live devices, and it may reduce the amount of additional coordination metadata that must be stored outside the MLS state machine itself. Exploring this redesign remains future work, since the current report focuses on a working and operationally simple master-device mechanism rather than on minimizing auxiliary coordination state.

7 Lessons Learned

The most important lesson from this project is that replacing a cryptographic primitive in a real system is not mainly a cryptography task. It is a distributed-systems task. Once the Rust/OpenMLS module was available in the browser, the difficult parts were not the MLS APIs themselves, but the surrounding requirements: ordering, persistence, recovery, availability of encrypted state after rekey, and integration with the existing NDN publication path.

A second lesson is that decentralized systems still need explicit coordination. MLS provides a well-defined group state machine, but the surrounding application must still decide who is allowed to advance that state machine and under what ordering rule. In this project, that issue became most visible in the owner multi-device case. Even though the workspace is decentralized, allowing several owner devices to generate and merge membership-changing commits concurrently would make global ordering ambiguous and could cause local state divergence. The final system therefore introduced a master owner device as the active commit sequencer. More generally, the lesson is that decentralization does not eliminate the need for coordination; it only changes where that coordination must be expressed.

A third lesson is that persistent state is not optional in a practical browser-based MLS deployment. A state machine that cannot survive refresh or restart is not sufficient for a collaborative system. Restoration must include not only visible group membership, but also the storage state, signing material, and enough local metadata to resume participation without rebuilding the group from scratch.

A fourth lesson is that protocol specifications and real deployments leave different problems unsolved. MLS is detailed about cryptographic group evolution, but it does not by itself tell an application how to persist state, republish encrypted snapshots after rekey, coordinate multiple devices that share administrative authority, or recover from partially inconsistent browser-local state. Those mechanisms are not outside the problem; they are part of what it means to deploy MLS in a real collaborative application.

Finally, I learned that recovery paths should be treated as part of the design rather than as debugging afterthoughts. In a browser-based collaborative system, local state can become stale, restarts can interrupt protocol progress, and users can leave and rejoin from different devices. A robust design therefore needs explicit reset, rejoin, and republish mechanisms from the beginning. The final system works not only because it rotates keys correctly, but because it also provides operational ways for the application to recover when the ideal protocol path is interrupted.

8 Conclusion

This project replaced Ownly's static PSK+DSK-derived workspace key with an MLS-managed group state built on OpenMLS, while preserving the surrounding NDN application architecture. The final system integrates Rust/WebAssembly OpenMLS state machines into the existing TypeScript and Go/WebAssembly codebase, transports KeyPackage, Welcome, and Commit objects over the existing SVS/blob path, persists MLS state across restarts, and exports one shared workspace key per epoch into Ownly's existing AES-GCM encryption path.

The evaluation shows that the resulting system is operational in the target browser-based deployment model. Member addition, removal, multi-device participation, restart recovery, and group-wide MLS reset all worked in the tested deployment. The timing results further show that once the required MLS objects are available locally, the MLS state transitions themselves are relatively inexpensive. The larger costs come from the surrounding system: bootstrap, DSK acquisition, publication delay variance, and encrypted-state recovery after key rotation.

The broader conclusion is that integrating MLS into a decentralized collaborative system is feasible, but it is not just a matter of swapping one cryptographic primitive for another. The protocol solves the core group key-evolution problem, but a practical

deployment still needs application-level policies for sequencing, persistence, and recovery. In Ownly, those policies turned out to be just as important as the MLS state machine itself.

A Selected Implementation Snippets

This appendix includes a few short excerpts from the final prototype. They focus on the parts of the implementation that are specific to the MLS integration rather than on generic application plumbing.

A.1 Transporting MLS Control Messages over SVS

Ownly does not replace its existing SVS publication path with native MLS application ciphertexts. Instead, it carries references to MLS KeyPackage, Welcome, and Commit blobs over the existing synchronization layer. The control message format and one publication wrapper are shown below.

```
type Message struct {
    MlsKeyPackage *MlsBlobRef `tlv:"0xD8"`
    MlsCommit *MlsBlobRef `tlv:"0xDA"`
    MlsWelcome *MlsBlobRef `tlv:"0xDC"`
}

type MlsBlobRef struct {
    Invitee string `tlv:"0x5A2"`
    BlobName string `tlv:"0x5A4"`
    SessionId string `tlv:"0x5A6"`
}

"pub_mls_kp_ref": jsutil.AsyncFunc(func(this js.Value, p []js.Value)
       (any, error) {
    pub := &tlv.Message{
        MlsKeyPackage: &tlv.MlsBlobRef{
            Invitee: p[0].String(),
            BlobName: p[1].String(),
            SessionId: p[2].String(),
        },
    }
    name, state, err := alo.Publish(pub.Encode())
    if err != nil {
        return nil, err
    }
    jsutil.Await(persistState.Invoke(jsutil.SliceToJsArray(state.Join
          ())))
    return js.ValueOf(name.String()), nil
})
```

A.2 Account-Level Authorization and Device-Level MLS Membership

Ownly authorizes workspace membership at the account level, but each physical device is represented as a separate MLS leaf. The TypeScript layer encodes an MLS identity as `accountId/deviceId`. During member addition, the owner checks the account component against the shared invitation map before calling OpenMLS.

```
export function parseMlsIdentity(identity: string):
      ParsedMlsIdentity {
  const normalized = normalizeAccountId(identity.trim());
  const idx = normalized.lastIndexOf('/');
  if (idx <= 0) {
    throw new Error(`Invalid MLS identity: ${identity}`);
  }
  const deviceId = normalized.slice(idx + 1);
  if (!UUID_RE.test(deviceId)) {
    throw new Error(`Invalid MLS identity: ${identity}`);
  }
  return {
    accountId: normalized.slice(0, idx),
    deviceId,
  };
}

export function accountIdentityPrefix(accountId: string): string {
  return `${normalizeAccountId(accountId)}/`;
}

const inviteeIdentity = await this.keyPackageIdentity(kp);
const identity = parseMlsIdentity(inviteeIdentity);
if (!this.inviteeProfiles.has(identity.accountId)) {
  throw new Error(`Invitee ${identity.accountId} is not authorized`);

}

const { commit, welcome } = group.addMembers([kp]);
group.mergePendingCommit();
```

Removal works in the other direction. The UI removes a logical account, but the MLS group may contain several device leaves for that account. The TypeScript layer asks the Rust/OpenMLS wrapper for all member indexes whose credentials start with the account prefix, and then removes those leaves in one Commit.

```
const indexes = group.memberIndexesByIdentityPrefix(
  encoder.encode(accountIdentityPrefix(name)));
if (!indexes.length) {
  throw new Error(`Member ${name} not found in MLS group`);
}
const { commit } = group.removeMembers(indexes);
group.mergePendingCommit();
```

The prefix search is implemented in the Rust/OpenMLS wrapper because the TypeScript layer cannot inspect OpenMLS group leaves directly. This helper decodes each member credential, strips the account prefix, and returns the leaf indexes for all devices under that account.

```
pub fn member_indexes_by_identity_prefix(&self, identity_prefix: &[
      u8]) -> Box<[u32]> {
    self.group
        .members()
        .filter(|m| {
            if m.credential.credential_type() != CredentialType::Basic
                  {
                return false;
            }
            let identity = match decode_workspace_credential(
                m.credential.serialized_content()) {
                Ok((identity, _workspace_cert)) => identity,
                Err(_) => return false,
            };
            let Some(device_id) = identity.strip_prefix(
                  identity_prefix) else {
                return false;
            };
            !device_id.is_empty() && !device_id.contains(&b'/')
        })
        .map(|m| m.index.u32())
        .collect::<Vec<_>>()
        .into_boxed_slice()
}
```

A.3 Directed MLS Reset

MLS reset was implemented as an owner-directed control path. Non-owner devices compute a directed reset-request name under the workspace root, and the trust schema authorizes only owner certificates to answer that request. The master owner device then clears local MLS state, publishes a reset marker, and reboots the owner-side MLS group.

```
#mls_rst_req:
  #owner/wksp/"root"/"32=MLS_RST_REQ"/#owner/_/_/#requester/_ <= #
        owner_cert
```

```
private mlsResetReqName(responder: string, requestId: string):
      string {
  return `${utils.normalizePath(this.api.group)}/root/32=MLS_RST_REQ
        ` +
    `${utils.normalizePath(responder)}/${requestId}` +
    `${utils.normalizePath(this.currentDeviceIdentity())}`;
}

public async resetGroupMlsState(): Promise<void> {
  await this.resetLocalMlsState('owner-triggered group reset');
  const resetBlob = await this.provider.publishBlob(
    `mls-reset-${Date.now()}`, new TextEncoder().encode('reset'));
  await this.provider.svs.pub_mls_commit_ref(
    MLS_RESET_SENTINEL, resetBlob, MLS_PREJOIN_SESSION_ID);
  await this.bootstrapOwnerMls();
  await this.notifyOwnerSessionAdvanced(this.currentMlsSessionId());
}
```

References

[1] R. Barnes, B. Beurdouche, R. Robert, J. Millican, E. Omara, and K. Cohn-Gordon, "The messaging layer security (mls) protocol," RFC Editor, Tech. Rep. RFC 9420, Jul. 2023. [Online]. Available: https://www.rfc-editor.org/info/rfc9420

[2] OpenMLS Authors, "OpenMLS: Rust implementation of the messaging layer security (mls) protocol," https://github.com/openmls/openmls, 2026, gitHub repository, accessed 2026-06-03.