Introduction

OpenMLS is a Rust implementation of the Messaging Layer Security (MLS) protocol, as specified in RFC 9420. OpenMLS provides a high-level API to create and manage MLS groups. It supports basic ciphersuites and an interchangeable cryptographic provider, key store, and random number generator.

This book provides guidance on using OpenMLS and its `MlsGroup` API to perform basic group operations, illustrated with examples.

Supported ciphersuites

- MLS_128_DHKEMX25519_AES128GCM_SHA256_Ed25519 (MTI)
- MLS_128_DHKEMP256_AES128GCM_SHA256_P256
- MLS_128_DHKEMX25519_CHACHA20POLY1305_SHA256_Ed25519

Supported platforms

OpenMLS is built and tested on the Github CI for the following rust targets.

- x86_64-unknown-linux-gnu
- i686-unknown-linux-gnu
- x86_64-pc-windows-msvc
- i686-pc-windows-msvc
- x86_64-apple-darwin

Unsupported, but built on CI

The Github CI also builds (but doesn't test) the following rust targets.

- aarch64-apple-darwin
- aarch64-unknown-linux-gnu
- aarch64-linux-android
- aarch64-apple-ios
- aarch64-apple-ios-sim
- wasm32-unknown-unknown
- armv7-linux-androideabi
- x86_64-linux-android
- i686-linux-android

OpenMLS supports 32 bit platforms and above.

Cryptography Dependencies

OpenMLS does not implement its own cryptographic primitives. Instead, it relies on existing implementations of the cryptographic primitives used by MLS. There are two different cryptography providers implemented right now. But consumers can bring their own implementation. See traits for more details.

Working on OpenMLS

For more details when working on OpenMLS itself please see the Developer.md.

Maintenance & Support

OpenMLS is maintained and developed by Phoenix R&D and Cryspen.

Acknowledgements

Zulip graciously provides the OpenMLS community with a "Zulip Cloud Standard" tier Zulip instance.

User manual

The user manual describes how to use the different parts of the OpenMLS API.

Prerequisites

Most operations in OpenMLS require a `provider` object that provides all required cryptographic algorithms via the `OpenMlsCryptoProvider` trait. Currently, there are two implementations available:

- one through the openmls_rust_crypto crate.
- one through the openmls_libcrux_crypto crate.

Thus, you can create the `provider` object for the following examples using ...

```
let provider: OpenMlsRustCrypto = OpenMlsRustCrypto::default();
```

Credentials

MLS relies on credentials to encode the identity of clients in the context of a group. There are different types of credentials, with the OpenMLS library currently only supporting the `BasicCredential` credential type (see below). Credentials are used to authenticate messages by the owner in the context of a group. Note that the link between the credential and its signature keys depends on the credential type. For example, the link between the `BasicCredential` 's and its keys is not defined by MLS.

A credential is always embedded in a leaf node, which is ultimately used to represent a client in a group and signed by the private key corresponding to the signature public key of the leaf node. Clients can decide to use the same credential in multiple leaf nodes (and thus multiple groups) or to use distinct credentials per group.

The binding between a given credential and owning client's identity is, in turn, authenticated by the Authentication Service, an abstract authentication layer defined by the MLS architecture document. Note that the implementation of the Authentication Service and, thus, the details of how the binding is authenticated are not specified by MLS.

Creating and using credentials

OpenMLS allows clients to create `Credentials` . A `BasicCredential` , currently the only credential type supported by OpenMLS, consists only of the `identity` . Thus, to create a fresh `Credential` , the following inputs are required:

- `identity: Vec<u8>` : An octet string that uniquely identifies the client.
- `credential_type: CredentialType` : The type of the credential, in this case `CredentialType::Basic` .

```
let credential = BasicCredential::new(identity);
```

After creating the credential bundle, clients should create keys for it. OpenMLS provides a simple implementation of `BasicCredential` for tests and to demonstrate how to use credentials.

```
let signature_keys = SignatureKeyPair::new(signature_algorithm).unwrap();
signature_keys.store(provider.storage()).unwrap();
```

All functions and structs related to credentials can be found in the `credentials` module.

Key packages

To enable the asynchronous establishment of groups through pre-publishing key material, as well as to represent clients in the group, MLS relies on key packages. Key packages hold several pieces of information:

- a public HPKE encryption key to enable MLS' basic group key distribution feature
- the lifetime throughout which the key package is valid
- information about the client's capabilities (i.e., which features of MLS it supports)
- any extension that the client wants to include
- one of the client's credentials, as well as a signature over the whole key package using the private key corresponding to the credential's signature public key

Creating key packages

Before clients can communicate with each other using OpenMLS, they need to generate key packages and publish them with the Delivery Service. Clients can generate an arbitrary number of key packages ahead of time.

Clients keep the private key material corresponding to a key package locally in the key store and fetch it from there when a key package was used to add them to a new group.

Clients need to choose a few parameters to create a `KeyPackageBundle`:

- `ciphersuites: &[CiphersuiteName]` : A list of ciphersuites supported by the client.
- `extensions: Vec<Extensions>` : A list of supported extensions.

Clients must specify at least one ciphersuite and not advertise ciphersuites they do not support.

Clients should specify all extensions they support. See the documentation of extensions for more details.

```
// Create the key package
KeyPackage::builder()
    .key_package_extensions(extensions)
    .build(ciphersuite, provider, signer, credential_with_key)
    .unwrap()
```

This will also store the private key for the key package in the key store.

All functions and structs related to key packages can be found in the `key_packages` module.

Group configuration

Two very similar structs can help configure groups upon their creation: `MlsGroupJoinConfig` and `MlsGroupCreateConfig`.

`MlsGroupJoinConfig` contains the following runtime-relevant configuration options for an `MlsGroup` and can be set on a per-client basis when a group is joined.

Name	Type	Explanation
`wire_format_policy`	`WireFormatPolicy`	Defines the wire format policy for outgoing and incoming handshake messages.
`padding_size`	`usize`	Size of padding in bytes. The default is 0.
`max_past_epochs`	`usize`	Maximum number of past epochs for which application messages can be decrypted. The default is 0.
`number_of_resumption_psks`	`usize`	Number of resumption psks to keep. The default is 0.
`use_ratchet_tree_extension`	`bool`	Flag indicating the Ratchet Tree

Name	Type	Explanation
		Extension should be used. The default is `false`.
`sender_ratchet_configuration`	`SenderRatchetConfiguration`	Sender ratchet configuration.

`MlsGroupCreateConfig` contains an `MlsGroupJoinConfig`, as well as a few additional parameters that are part of the group state that is agreed-upon by all group members. It can be set at the time of a group's creation and contains the following additional configuration options.

Name	Type	Explanation
`group_context_extensions`	`Extensions`	Optional group-level extensions, e.g. `RequiredCapabilitiesExtension`.
`capabilities .`	`Capabilities`	Lists the capabilities of the group's creator.
`leaf_extensions .`	`Extensions`	Extensions to be included in the group creator's leaf

Both ways of group configurations can be specified by using the struct's builder pattern, or choosing their default values. The default value contains safe values for all parameters and is suitable for scenarios without particular requirements.

Example join configuration:

```
let mls_group_config = MlsGroupJoinConfig::builder()
    .padding_size(100)
    .sender_ratchet_configuration(SenderRatchetConfiguration::new(
        10,   // out_of_order_tolerance
        2000, // maximum_forward_distance
    ))
    .use_ratchet_tree_extension(true)
    .build();
```

Example create configuration:

```
    let mls_group_create_config = MlsGroupCreateConfig::builder()
        .padding_size(100)
        .sender_ratchet_configuration(SenderRatchetConfiguration::new(
            10,   // out_of_order_tolerance
            2000, // maximum_forward_distance
        ))

.with_group_context_extensions(Extensions::single(Extension::ExternalSenders(vec![
            ExternalSender::new(
                ds_credential_with_key.signature_key.clone(),
                ds_credential_with_key.credential.clone(),
            ),
        ])))
        .expect("error adding external senders extension to group context extensions")
        .ciphersuite(ciphersuite)
        // we need to specify the non-default extension here
        .capabilities(Capabilities::new(
            None, // Defaults to the group's protocol version
            None, // Defaults to the group's ciphersuite
            Some(&[ExtensionType::Unknown(0xff00)]),
            None, // Defaults to all basic extension types
            Some(&[CredentialType::Basic]),
        ))
        // Example leaf extension
        .with_leaf_node_extensions(Extensions::single(Extension::Unknown(
            0xff00,
            UnknownExtension(vec![0, 1, 2, 3]),
        )))
        .expect("failed to configure leaf extensions")
        .use_ratchet_tree_extension(true)
        .build();
```

Unknown extensions

Some extensions carry data, but don't alter the behaviour of the protocol (e.g. the application_id

extension). OpenMLS allows the use of arbitrary such extensions in the group context, key packages and leaf nodes. Such extensions can be instantiated and retrieved through the use of the `UnknownExtension` struct and the `ExtensionType::Unknown` extension type. Such "unknown" extensions are handled transparently by OpenMLS, but can be used by the application, e.g. to have a group agree on pieces of data.

Creating groups

There are two ways to create a group: Either by building an `MlsGroup` directly, or by using an `MlsGroupCreateConfig`. The former is slightly simpler, while the latter allows the creating of multiple groups using the same configuration. See Group configuration for more details on group parameters.

In addition to the group configuration, the client should define all supported and required extensions for the group. The negotiation mechanism for extension in MLS consists in setting an initial list of extensions at group creation time and choosing key packages of subsequent new members accordingly.

In practice, the supported and required extensions are set by adding them to the initial `KeyPackage` of the creator:

```
// Create the key package
KeyPackage::builder()
    .key_package_extensions(extensions)
    .build(ciphersuite, provider, signer, credential_with_key)
    .unwrap()
```

After that, the group can be created either using a config:

```
let mut alice_group = MlsGroup::new(
    alice_provider,
    &alice_signature_keys,
    &mls_group_create_config,
    alice_credential.clone(),
)
.expect("An unexpected error occurred.");
```

... or using the builder pattern:

```
let mut alice_group = MlsGroup::builder()
    .padding_size(100)
    .sender_ratchet_configuration(SenderRatchetConfiguration::new(
        10,   // out_of_order_tolerance
        2000, // maximum_forward_distance
    ))
    .ciphersuite(ciphersuite)
    .use_ratchet_tree_extension(true)
    .build(
        alice_provider,
        &alice_signature_keys,
        alice_credential.clone(),
    )
    .expect("An unexpected error occurred.");
```

Note: Every group is assigned a random group ID during creation. The group ID cannot be changed and remains immutable throughout the group's lifetime. Choosing it randomly makes sure that the group ID doesn't collide with any other group ID in the same system.

If someone else already gave you a group ID, e.g., a provider server, you can also create a group using a specific group ID:

```
// Some specific group ID generated by someone else.
let group_id = GroupId::from_slice(b"123e4567e89b");

let mut alice_group = MlsGroup::new_with_group_id(
    alice_provider,
    &alice_signature_keys,
    &mls_group_create_config,
    group_id,
    alice_credential.clone(),
)
.expect("An unexpected error occurred.");
```

The Builder provides methods for setting required capabilities and external senders. The information passed into these lands in the group context, in the form of extensions. Should the user

want to add further extensions, they can use the `with_group_context_extensions` method:

```
        // we are adding an external senders list as an example.
        let extensions =
            Extensions::from_vec(vec!
[Extension::ExternalSenders(external_senders_list)])
                .expect("failed to create extensions list");

        let mut alice_group = MlsGroup::builder()
            .padding_size(100)
            .sender_ratchet_configuration(SenderRatchetConfiguration::new(
                10,   // out_of_order_tolerance
                2000, // maximum_forward_distance
            ))
            .with_group_context_extensions(extensions) // NB: the builder method
returns a Result
            .expect("failed to apply group context extensions")
            .use_ratchet_tree_extension(true)
            .build(
                alice_provider,
                &alice_signature_keys,
                alice_credential.clone(),
            )
            .expect("An unexpected error occurred.");
```

Join a group from a Welcome message

To join a group from a `Welcome` message, a new `MlsGroup` can be instantiated from the `MlsMessageIn` message containing the `Welcome` and an `MlsGroupJoinConfig` (see Group configuration for more details). This is a two-step process: a `StagedWelcome` is constructed from the `Welcome` and can then be turned into an `MlsGroup`. If the group configuration does not use the ratchet tree extension, the ratchet tree needs to be provided.

```
    let staged_join =
        StagedWelcome::new_from_welcome(bob_provider, &mls_group_config, welcome,
None)
            .expect("Error constructing staged join");
    let mut bob_group = staged_join
        .into_group(bob_provider)
        .expect("Error joining group from StagedWelcome");
```

The reason for this two-phase process is to allow the recipient of a `Welcome` to inspect the message, e.g. to determine the identity of the sender.

Pay attention not to forward a Welcome message to a client before its associated commit has been accepted by the Delivery Service. Otherwise, you would end up with an invalid MLS group instance.

Examining a welcome message

When a client is invited to join a group, the application can allow the client to decide whether or not to join the group. In order to determine whether to join the group, the application can inspect information provided in the welcome message, such as who invited them, who else is in the group, what extensions are available, and more. If the application decides not to join the group, the welcome must be discarded to ensure that the local state is cleaned up.

After receiving a `MlsMessageIn` from the delivery service, the first step is to extract the

`MlsMessageBodyIn`, and determine whether it is a welcome message.

```
    let welcome = match welcome.extract() {
        MlsMessageBodyIn::Welcome(welcome) => welcome,
        _ => unimplemented!("Handle other message types"),
    };
```

The next step is to process the `Welcome`. This removes the consumed `KeyPackage` from the `StorageProvider`, unless it is a last resort `KeyPackage`.

```
    let join_config = MlsGroupJoinConfig::default();
    // This deletes the keys used to decrypt the welcome, except if it is a last
resort key
    // package.
    let processed_welcome = ProcessedWelcome::new_from_welcome(bob_provider,
&join_config, welcome)
        .expect("Error constructing processed welcome");
```

At this stage, there are some more pieces of information in the `ProcessedWelcome` that could be useful to the application. For example, it can be useful to check which extensions are available. However, the pieces of information that are retrieved from the `ProcessedWelcome` are unverified, and verified values are only available from the `StagedWelcome` that is produced in the next step.

```
    // unverified pre-shared keys (`&[PreSharedKeyId]`)
    let _unverified_psks = processed_welcome.psks();

    // unverified group info (`VerifiableGroupInfo`)
    let unverified_group_info = processed_welcome.unverified_group_info();

    // From the unverified group info, the ciphersuite, group_id, and other
information
    // can be retrieved.
    let _ciphersuite = unverified_group_info.ciphersuite();
    let _group_id = unverified_group_info.group_id();
    let _epoch = unverified_group_info.epoch();

    // Can also retrieve any available extensions
    let extensions = unverified_group_info.extensions();

    // Retrieving the ratchet tree extension
    let ratchet_tree_extension = extensions
        .ratchet_tree()
        .expect("No ratchet tree extension");
    // The (unverified) ratchet tree itself can also be inspected
    let _ratchet_tree = ratchet_tree_extension.ratchet_tree();
```

The next step is to stage the `ProcessedWelcome`.

```
    let staged_welcome: StagedWelcome = processed_welcome
        .into_staged_welcome(bob_provider, None)
        .expect("Error constructing staged welcome");
```

Then, more information about the welcome message's sender, such as the credential, signature public key, and encryption public key can also be individually inspected. The welcome message sender's credential can be validated at this stage.

```
let welcome_sender: &LeafNode = staged_welcome
    .welcome_sender()
    .expect("Welcome sender could not be retrieved");

// Inspect sender's credential...
let _credential = welcome_sender.credential();
// Inspect sender's signature public key...
let _signature_key = welcome_sender.signature_key();
```

Additionally, some information about the other group members is made available, e.g. credentials and signature public keys for credential validation.

```
// Inspect the group members
for member in staged_welcome.members() {
    // leaf node index
    let _leaf_node_index = member.index;
    // credential
    let _credential = member.credential;
    // encryption public key
    let _encryption_key = member.encryption_key;
    // signature public key
    let _signature_key = member.signature_key;
}
```

Lastly, the `GroupContext` contains several other useful pieces of information, including the protocol version, the extensions enabled on the group, and the required extension, proposal, and credential types.

```
// Inspect group context...
let group_context = staged_welcome.group_context();

// inspect protocol version...
let _protocol_version = group_context.protocol_version();
// Inspect ciphersuite...
let _ciphersuite = group_context.ciphersuite();
// Inspect extensions...
let extensions: &Extensions = group_context.extensions();

// Can check which extensions are enabled
let _has_ratchet_extension = extensions.ratchet_tree().is_some();

// Inspect required capabilities...
if let Some(capabilities) = group_context.required_capabilities() {
    // Inspect required extension types...
    let _extension_types: &[ExtensionType] = capabilities.extension_types();
    // Inspect required proposal types...
    let _proposal_types: &[ProposalType] = capabilities.proposal_types();
    // Inspect required credential types...
    let _credential_types: &[CredentialType] = capabilities.credential_types();
}
// Additional information from the `GroupContext`
let _group_id = group_context.group_id();
let _epoch = group_context.epoch();
let _tree_hash = group_context.tree_hash();
let _confirmed_transcript_hash = group_context.confirmed_transcript_hash();
```

Join a group with an external commit

To join a group with an external commit message, a new `MlsGroup` can be instantiated directly from the `GroupInfo`. The `GroupInfo`/Ratchet Tree should be shared over a secure channel. If the RatchetTree extension is not included in the `GroupInfo` as a `GroupInfoExtension`, then the ratchet tree needs to be provided.

The `GroupInfo` can be obtained either from a call to `export_group_info` from the `MlsGroup`:

```
let (mls_message_out, welcome, group_info) = alice_group
    .add_members(
        alice_provider,
        &alice_signature_keys,
        core::slice::from_ref(bob_key_package.key_package()),
    )
    .expect("Could not add members.");
```

Or from a call to a function that results in a staged commit:

```
let verifiable_group_info = alice_group
    .export_group_info(alice_provider.crypto(), &alice_signature_keys, true)
    .expect("Cannot export group info")
    .into_verifiable_group_info()
    .expect("Could not get group info");
```

External commits can be created using a builder pattern via `MlsGroup::external_commit_builder()`. The `ExternalCommitBuilder` provides more options than `join_by_external` in that it allows the inclusion of SelfRemove or PSK proposals. After its first stage, the `ExternalCommitBuilder` turns into a regular `CommitBuilder`. As external commits come with a few restrictions relative to regular commits, not all `CommitBuilder` capabilities are exposed for external commits. Also, instead of `stage_commit` this `CommitBuilder` requires a call to `finalize` before it returns the new `MlsGroup`, as well as a `CommitMessageBundle` containing the external

commit, as well as a potential `GroupInfo`.

```
let (mut bob_group, commit_message_bundle) = MlsGroup::external_commit_builder()
    .with_ratchet_tree(tree_option.into())
    .with_config(join_group_config.clone())
    .with_aad(AAD.to_vec())
    .build_group(
        bob_provider,
        verifiable_group_info,
        bob_credential_with_key.clone(),
    )
    .expect("error building group")
    .leaf_node_parameters(leaf_node_parameters)
    .load_psks(bob_provider.storage())
    .expect("error loading psks")
    .build(
        bob_provider.rand(),
        bob_provider.crypto(),
        &bob_signer,
        |_| true,
    )
    .expect("error building external commit")
    .finalize(bob_provider)
    .expect("error finalizing external commit");
```

The resulting external commit message needs to be fanned out to the Delivery Service and accepted by the other members before merging this external commit.

Adding members to a group

Immediate operation

Members can be added to the group atomically with the `.add_members()` function. The application needs to fetch the corresponding key packages from every new member from the Delivery Service first.

```
let (mls_message_out, welcome, group_info) = alice_group
    .add_members(
        alice_provider,
        &alice_signature_keys,
        core::slice::from_ref(bob_key_package.key_package()),
    )
    .expect("Could not add members.");
```

The function returns the tuple `(MlsMessageOut, Welcome, Option<GroupInfo>)`. The `MlsMessageOut` contains a Commit message that needs to be fanned out to existing group members. The `Welcome` message must be sent to the newly added members, along the optional `GroupInfo` if it is available.

Users could also use the new `CommitBuilder` API, which would look like this:

```
let message_bundle = alice_group
    .commit_builder()
    .propose_adds(Some(bob_key_package.key_package().clone()))
    .load_psks(alice_provider.storage())
    .expect("error loading psks")
    .build(
        alice_provider.rand(),
        alice_provider.crypto(),
        &alice_signature_keys,
        |_proposal| true,
    )
    .expect("error validating data and building commit")
    .stage_commit(alice_provider)
    .expect("error staging commit");

let (mls_message_out, welcome, group_info) = message_bundle.into_contents();
```

Some notes on the arguments to the builder stages:

- The reason that the `KeyPackage` is wrapped in a `Some` is that `Option<KeyPackage>` implements `IntoIterator<Item = KeyPackage>`, which is the type bounds of that function. This means that the function also works with any iterator over `KeyPackage` items or a `Vec<KeyPackage>`.
- The closure is a predicate over `&QueuedProposal` and represents the policy of which proposals are deemed acceptable in the application.

This function returns a `CommitMessageBundle`, from which the `MlsMessageOut`, `Welcome` and `GroupInfo` can be extracted.

Adding members without update

The `.add_members_without_update()` function functions the same as the `.add_members()` function, except that it will only include an update to the sender's key material if the sender's proposal store

includes a proposal that requires a path. For a list of proposals and an indication whether they require a `path` (i.e. a key material update) see Section 17.4 of RFC 9420.

Not sending an update means that the sender will not achieve post-compromise security with this particular commit. However, not sending an update saves on performance both in terms of computation and bandwidth. Using `.add_members_without_update()` can thus be a useful option if the ciphersuite of the group features large public keys and/or expensive encryption operations.

Proposal

Members can also be added as a proposal (without the corresponding Commit message) by using the `.propose_add_member()` function:

```
let (mls_message_out, _proposal_ref) = alice_group
    .propose_add_member(
        alice_provider,
        &alice_signature_keys,
        bob_key_package.key_package(),
    )
    .expect("Could not create proposal to add Bob");
```

In this case, the function returns an `MlsMessageOut` that needs to be fanned out to existing group members.

External proposal

Parties outside the group can also make proposals to add themselves to the group with an external proposal. Since those proposals are crafted by outsiders, they are always plaintext messages.

```
    let proposal =
        JoinProposal::new::<<Provider as
openmls_traits::OpenMlsProvider>::StorageProvider>(
            bob_key_package.key_package().clone(),
            alice_group.group_id().clone(),
            alice_group.epoch(),
            &bob_signature_keys,
        )
        .expect("Could not create external Add proposal");
```

It is then up to the group members to validate the proposal and commit it. Note that in this scenario it is up to the application to define a proper authorization policy to grant the sender.

```
    let alice_processed_message = alice_group
        .process_message(
            alice_provider,
            proposal
                .into_protocol_message()
                .expect("Unexpected message type."),
        )
        .expect("Could not process message.");
    match alice_processed_message.into_content() {
        ProcessedMessageContent::ExternalJoinProposalMessage(proposal) => {
            alice_group
                .store_pending_proposal(alice_provider.storage(), *proposal)
                .unwrap();
            let (_commit, welcome, _group_info) = alice_group
                .commit_to_pending_proposals(alice_provider, &alice_signature_keys)
                .expect("Could not commit");
            assert_eq!(alice_group.members().count(), 1);
            alice_group
                .merge_pending_commit(alice_provider)
                .expect("Could not merge commit");
            assert_eq!(alice_group.members().count(), 2);

            let welcome: MlsMessageIn = welcome.expect("Welcome was not
returned").into();
            let welcome = welcome
                .into_welcome()
                .expect("expected the message to be a welcome message");

            let bob_group = StagedWelcome::new_from_welcome(
                bob_provider,
                mls_group_create_config.join_config(),
                welcome,
                None,
            )
            .expect("Bob could not stage the the group join")
            .into_group(bob_provider)
            .expect("Bob could not join the group");
            assert_eq!(bob_group.members().count(), 2);
        }
```

```
        _ => unreachable!(),
    }
```

Outside parties can also make proposals to add other members as long as they are registered as part of the `ExternalSendersExtension` extension. Since those proposals are crafted by outsiders, they are always public messages.

```
let proposal = ExternalProposal::new_add::<Provider>(
    bob_key_package.key_package().clone(),
    alice_group.group_id().clone(),
    alice_group.epoch(),
    &ds_signature_keys,
    SenderExtensionIndex::new(0),
)
.expect("Could not create external Add proposal");
```

It is then up to one of the group members to process the proposal and commit it.

```
let alice_processed_message = alice_group
    .process_message(
        alice_provider,
        proposal
            .into_protocol_message()
            .expect("Unexpected message type."),
    )
    .expect("Could not process message.");
match alice_processed_message.into_content() {
    ProcessedMessageContent::ProposalMessage(proposal) => {
        alice_group
            .store_pending_proposal(alice_provider.storage(), *proposal)
            .unwrap();
        assert_eq!(alice_group.members().count(), 2);
        alice_group
            .commit_to_pending_proposals(alice_provider, &alice_signature_keys)
            .expect("Could not commit");
        alice_group
            .merge_pending_commit(alice_provider)
            .expect("Could not merge commit");
        assert_eq!(alice_group.members().count(), 1);
    }
    _ => unreachable!(),
}
```

Removing members from a group

Immediate operation

Members can be removed from the group atomically with the `.remove_members()` function, which takes the `KeyPackageRef` of group member as input. References to the `KeyPackage`s of group members can be obtained using the `.members()` function, from which one can in turn compute the `KeyPackageRef` using their `.hash_ref()` function.

```
let (mls_message_out, welcome_option, _group_info) = charlie_group
    .remove_members(
        charlie_provider,
        &charlie_signature_keys,
        &[bob_member.index],
    )
    .expect("Could not remove Bob from group.");
```

The function returns the tuple `(MlsMessageOut, Option<Welcome>)`. The `MlsMessageOut` contains a Commit message that needs to be fanned out to existing group members. Even though members were removed in this operation, the Commit message could potentially also cover Add Proposals previously received in the epoch. Therefore the function can also optionally return a `Welcome` message. The `Welcome` message must be sent to the newly added members.

Proposal

Members can also be removed as a proposal (without the corresponding Commit message) by using the `.propose_remove_member()` function:

```
let (mls_message_out, _proposal_ref) = alice_group
    .propose_remove_member(
        alice_provider,
        &alice_signature_keys,
        charlie_group.own_leaf_index(),
    )
    .expect("Could not create proposal to remove Charlie.");
```

In this case, the function returns an `MlsMessageOut` that needs to be fanned out to existing group members.

Getting removed from a group

A member is removed from a group if another member commits to a remove proposal targeting the member's leaf. Once the to-be-removed member merges that commit via `merge_staged_commit()`, all other proposals in that commit will still be applied, but the group will be marked as inactive afterward. The group remains usable, e.g., to examine the membership list after the final commit was processed, but it won't be possible to create or process new messages.

```
    if let ProcessedMessageContent::StagedCommitMessage(staged_commit) =
        bob_processed_message.into_content()
    {
        let remove_proposal = staged_commit
            .remove_proposals()
            .next()
            .expect("An unexpected error occurred.");

        // We construct a RemoveOperation enum to help us interpret the remove
operation
        let remove_operation = RemoveOperation::new(remove_proposal, &bob_group)
            .expect("An unexpected Error occurred.");

        match remove_operation {
            RemoveOperation::WeLeft => unreachable!(),
            // We expect this variant, since Bob was removed by Charlie
            RemoveOperation::WeWereRemovedBy(member) => {
                assert!(matches!(member, Sender::Member(member) if member ==
charlies_leaf_index));
            }
            RemoveOperation::TheyLeft(_) => unreachable!(),
            RemoveOperation::TheyWereRemovedBy(_) => unreachable!(),
            RemoveOperation::WeRemovedThem(_) => unreachable!(),
        }

        // Merge staged Commit
        bob_group
            .merge_staged_commit(bob_provider, *staged_commit)
            .expect("Error merging staged commit.");
    } else {
        unreachable!("Expected a StagedCommit.");
    }

    // Check we didn't receive a Welcome message
    assert!(welcome_option.is_none());

    // Check that Bob's group is no longer active
    assert!(!bob_group.is_active());
    let members = bob_group.members().collect::<Vec<Member>>();
```

```
assert_eq!(members.len(), 2);
let credential0 = members[0].credential.serialized_content();
let credential1 = members[1].credential.serialized_content();
assert_eq!(credential0, b"Alice");
assert_eq!(credential1, b"Charlie");
```

External Proposal

Parties outside the group can also make proposals to remove members as long as they are registered as part of the `ExternalSendersExtension` extension. Since those proposals are crafted by outsiders, they are always public messages.

```
let proposal = ExternalProposal::new_remove::<Provider>(
    bob_index,
    alice_group.group_id().clone(),
    alice_group.epoch(),
    &ds_signature_keys,
    SenderExtensionIndex::new(0),
)
.expect("Could not create external Remove proposal");
```

It is then up to one of the group members to process the proposal and commit it.

```
let alice_processed_message = alice_group
    .process_message(
        alice_provider,
        proposal
            .into_protocol_message()
            .expect("Unexpected message type."),
    )
    .expect("Could not process message.");
match alice_processed_message.into_content() {
    ProcessedMessageContent::ProposalMessage(proposal) => {
        alice_group
            .store_pending_proposal(alice_provider.storage(), *proposal)
            .unwrap();
        assert_eq!(alice_group.members().count(), 2);
        alice_group
            .commit_to_pending_proposals(alice_provider, &alice_signature_keys)
            .expect("Could not commit");
        alice_group
            .merge_pending_commit(alice_provider)
            .expect("Could not merge commit");
        assert_eq!(alice_group.members().count(), 1);
    }
    _ => unreachable!(),
}
```

Updating own leaf node

Immediate operation

Members can update their own leaf node atomically with the `.self_update()` function. By default, only the HPKE encryption key is updated. The application can however also provide more parameters like a new credential, capabilities and extensions using the `LeafNodeParameters` struct.

```
let (mls_message_out, welcome_option, _group_info) = bob_group
    .self_update(
        bob_provider,
        &bob_signature_keys,
        LeafNodeParameters::default(),
    )
    .expect("Could not update own key package.")
    .into_contents();
```

The function returns a `CommitMessageBundle`, which consists of the Commit message that needs to be fanned out to existing group members. Even though the member updates its own leaf node only in this operation, the Commit message could potentially also cover Add Proposals that were previously received in the epoch. Therefore the `CommitMessagBundle` can also contain a `Welcome` message. The `Welcome` message must be sent to the newly added members.

Members can use the `.self_update_with_new_signer()` function to also update the `Signer` used to sign future MLS messages.

```
let new_signer_bundle = NewSignerBundle {
    signer: &alice_new_signature_keys,
    credential_with_key: alice_new_credential,
};

let message_bundle = alice_group
    .self_update_with_new_signer(
        alice_provider,
        &alice_old_signature_keys,
        new_signer_bundle,
        LeafNodeParameters::default(),
    )
    .unwrap();

let (mls_message_out, welcome, group_info) = message_bundle.into_contents();
```

When constructing the `NewSignerBundle`, the `Signer` must match the public key and credential in the `CredentialWithKey`. When using `self_update_with_new_signer`, `LeafNodeParameters` may not contain a `CredentialWithKey`.

Proposal

Members can also update their leaf node as a proposal (without the corresponding Commit message) by using the `.propose_self_update()` function. Just like with the `.self_update()` function, optional parameters can be set through `LeafNodeParameters`:

```
let (mls_message_out, _proposal_ref) = alice_group
    .propose_self_update(
        alice_provider,
        &alice_signature_keys,
        LeafNodeParameters::default(),
    )
    .expect("Could not create update proposal.");
```

In this case, the function returns an `MlsMessageOut` that needs to be fanned out to existing group members.

Using Additional Authenticated Data (AAD)

The Additional Authenticated Data (AAD) is a byte sequence that can be included in both private and public messages. By design, it is always authenticated (signed) but never encrypted. Its purpose is to contain data that can be inspected but not changed while a message is in transit.

Setting the AAD

Members can set the AAD by calling the `.set_aad()` function. The AAD will remain set until the next API call that successfully generates an `MlsMessageOut`. Until then, the AAD can be inspected with the `.aad()` function.

```
alice_group.set_aad(b"Additional Authenticated Data".to_vec());

assert_eq!(alice_group.aad(), b"Additional Authenticated Data");
```

Inspecting the AAD

Members can inspect the AAD of an incoming message once the message has been processed. The AAD can be accessed with the `.aad()` function of a `ProcessedMessage`.

```
let processed_message = bob_group
    .process_message(bob_provider, protocol_message)
    .expect("Could not process message.");

assert_eq!(processed_message.aad(), b"Additional Authenticated Data");
```

Leaving a group

Members can indicate to other group members that they wish to leave the group using the `leave_group()` function, which creates a remove proposal targeting the member's own leaf. The member can't create a Commit message that covers this proposal, as that would violate the Post-compromise Security guarantees of MLS because the member would know the epoch secrets of the next epoch.

```
let queued_message = bob_group
    .leave_group(bob_provider, &bob_signature_keys)
    .expect("Could not leave group");
```

After successfully sending the proposal to the DS for fanout, there is still the possibility that the remove proposal is not covered in the following commit. The member leaving the group thus has two options:

- tear down the local group state and ignore all subsequent messages for that group, or
- wait for the commit to come through and process it (see also Getting Removed).

For details on creating Remove Proposals, see Removing members from a group.

Custom proposals

OpenMLS allows the creation and use of application-defined proposals. To create such a proposal, the application needs to define a Proposal Type in such a way that its value doesn't collide with any Proposal Types defined in Section 17.4. of RFC 9420. If the proposal is meant to be used only inside of a particular application, the value of the Proposal Type is recommended to be in the range between `0xF000` and `0xFFFF`, as that range is reserved for private use.

Custom proposals can contain arbitrary octet-strings as defined by the application. Any policy decisions based on custom proposals will have to be made by the application, such as the decision to include a given custom proposal in a commit, or whether to accept a commit that includes one or more custom proposals. To decide the latter, applications can inspect the queued proposals in a `ProcessedMessageContent::StagedCommitMessage(staged_commit)`.

Example on how to use custom proposals:

```
// Define a custom proposal type
let custom_proposal_type = 0xFFFF;

// Define capabilities supporting the custom proposal type
let capabilities = Capabilities::new(
    None,
    None,
    None,
    Some(&[ProposalType::Custom(custom_proposal_type)]),
    None,
);

// Generate KeyPackage that signals support for the custom proposal type
let bob_key_package = KeyPackageBuilder::new()
    .leaf_node_capabilities(capabilities.clone())
    .build(
        ciphersuite,
        bob_provider,
        &bob_signer,
        bob_credential_with_key,
    )
    .unwrap();

// Create a group that supports the custom proposal type
let mut alice_group = MlsGroup::builder()
    .with_capabilities(capabilities.clone())
    .ciphersuite(ciphersuite)
    .build(alice_provider, &alice_signer, alice_credential_with_key)
    .unwrap();
```

```
    // Create a custom proposal based on an example payload and the custom
    // proposal type defined above
    let custom_proposal_payload = vec![0, 1, 2, 3];
    let custom_proposal =
        CustomProposal::new(custom_proposal_type, custom_proposal_payload.clone());

    let (custom_proposal_message, _proposal_ref) = alice_group
        .propose_custom_proposal_by_reference(
            alice_provider,
            &alice_signer,
            custom_proposal.clone(),
        )
        .unwrap();

    // Have bob process the custom proposal.
    let processed_message = bob_group
        .process_message(
            bob_provider,
            custom_proposal_message.into_protocol_message().unwrap(),
        )
        .unwrap();

    let ProcessedMessageContent::ProposalMessage(proposal) =
processed_message.into_content()
    else {
        panic!("Unexpected message type");
    };

    bob_group
        .store_pending_proposal(bob_provider.storage(), *proposal)
        .unwrap();

    // Commit to the proposal
    let (commit, _, _) = alice_group
        .commit_to_pending_proposals(alice_provider, &alice_signer)
        .unwrap();

    let processed_message = bob_group
        .process_message(bob_provider, commit.into_protocol_message().unwrap())
```

```
    .unwrap();

let staged_commit = match processed_message.into_content() {
    ProcessedMessageContent::StagedCommitMessage(staged_commit) => staged_commit,
    _ => panic!("Unexpected message type"),
};

// Check that the proposal is present in the staged commit
assert!(staged_commit.queued_proposals().any(|qp| {
    let Proposal::Custom(custom_proposal) = qp.proposal() else {
        return false;
    };
    custom_proposal.proposal_type() == custom_proposal_type
        && custom_proposal.payload() == custom_proposal_payload
}));
```

Creating application messages

Application messages are created from byte slices with the `.create_message()` function:

```
let message_alice = b"Hi, I'm Alice!";
let mls_message_out = alice_group
    .create_message(alice_provider, &alice_signature_keys, message_alice)
    .expect("Error creating application message.");
```

Note that the theoretical maximum length of application messages is 2^32 bytes. However, messages should be much shorter in practice unless the Delivery Service can cope with long messages.

The function returns an `MlsMessageOut` that needs to be sent to the Delivery Service for fanout to other group members. To guarantee the best possible Forward Secrecy, the key material used to encrypt messages is immediately discarded after encryption. This means that the message author cannot decrypt application messages. If access to the message's content is required after creating the message, a copy of the plaintext message should be kept by the application.

Committing to pending proposals

During an epoch, members can create proposals that are not immediately committed. These proposals are called "pending proposals". They will automatically be covered by any operation that creates a Commit message (like `.add_members()`, `.remove_members()`, etc.).

Some operations (like creating application messages) are not allowed as long as pending proposals exist for the current epoch. In that case, the application must first commit to the pending proposals by creating a Commit message that covers these proposals. This can be done with the `commit_to_pending_proposals()` function:

```
let (mls_message_out, welcome_option, _group_info) = alice_group
    .commit_to_pending_proposals(alice_provider, &alice_signature_keys)
    .expect("Could not commit to pending proposals.");
```

The function returns the tuple `(MlsMessageOut, Option<Welcome>)`. The `MlsMessageOut` contains a Commit message that needs to be fanned out to existing group members. If the Commit message also covers Add Proposals previously received in the epoch, a `Welcome` message is required to invite the new members. Therefore the function can also optionally return a `Welcome` message that must be sent to the newly added members.

Processing incoming messages

Processing of incoming messages happens in different phases:

Deserializing messages

Incoming messages can be deserialized from byte slices into an `MlsMessageIn`:

```
    let mls_message =
        MlsMessageIn::tls_deserialize_exact(bytes).expect("Could not deserialize
message.");
```

If the message is malformed, the function will fail with an error.

Processing messages in groups

In the next step, the message needs to be processed in the context of the corresponding group.

`MlsMessageIn` can carry all MLS messages, but only `PrivateMessageIn` and `PublicMessageIn` are processed in the context of a group. In OpenMLS these two message types are combined into a `ProtocolMessage enum`. There are 3 ways to extract the messages from an `MlsMessageIn`:

1. `MlsMessageIn.try_into_protocol_message()` returns a `Result<ProtocolMessage, ProtocolMessageError>`
2. `ProtocolMessage::try_from(m: MlsMessageIn)` returns a `Result<ProtocolMessage, ProtocolMessageError>`
3. `MlsMessageIn.extract()` returns an `MlsMessageBodyIn enum` that has two variants for

`PrivateMessageIn` and `PublicMessageIn`

`MlsGroup.process_message()` accepts either a `ProtocolMessage`, a `PrivateMessageIn`, or a `PublicMessageIn` and processes the message. `ProtocolMessage.group_id()` exposes the group ID that can help the application find the right group.

If the message was encrypted (i.e. if it was a `PrivateMessageIn`), it will be decrypted automatically. The processing performs all syntactic and semantic validation checks and verifies the message's signature. The function finally returns a `ProcessedMessage` object if all checks are successful.

```
let protocol_message: ProtocolMessage = mls_message
    .try_into_protocol_message()
    .expect("Expected a PublicMessage or a PrivateMessage");
let processed_message = bob_group
    .process_message(bob_provider, protocol_message)
    .expect("Could not process message.");
```

Interpreting the processed message

In the last step, the message is ready for inspection. The `ProcessedMessage` obtained in the previous step exposes header fields such as group ID, epoch, sender, and authenticated data. It also exposes the message's content. There are 3 different content types:

Application messages

Application messages simply return the original byte slice:

```
    if let ProcessedMessageContent::ApplicationMessage(application_message) =
        processed_message.into_content()
    {
        // Check the message
        assert_eq!(application_message.into_bytes(), b"Hi, I'm Alice!");
    }
```

Proposals

Standalone proposals are returned as a `QueuedProposal`, indicating that they are pending proposals. The proposal can be inspected through the `.proposal()` function. After inspection, applications should store the pending proposal in the proposal store of the group:

```
if let ProcessedMessageContent::ProposalMessage(staged_proposal) =
    charlie_processed_message.into_content()
{
    // In the case we received an Add Proposal
    if let Proposal::Add(add_proposal) = staged_proposal.proposal() {
        // Check that Bob was added
        assert_eq!(
            add_proposal.key_package().leaf_node().credential(),
            &bob_credential.credential
        );
    } else {
        panic!("Expected an AddProposal.");
    }

    // Check that Alice added Bob
    assert!(matches!(
        staged_proposal.sender(),
        Sender::Member(member) if *member == alice_group.own_leaf_index()
    ));
    // Store proposal
    charlie_group
        .store_pending_proposal(charlie_provider.storage(), *staged_proposal)
        .unwrap();
}
```

Rolling back proposals

Operations that add a proposal to the proposal store, will return its reference. This reference can be used to remove a proposal from the proposal store. This can be useful for example to roll back in case of errors.

```
let (_mls_message_out, proposal_ref) = alice_group
    .propose_add_member(
        alice_provider,
        &alice_signature_keys,
        bob_key_package.key_package(),
    )
    .expect("Could not create proposal to add Bob");
alice_group
    .remove_pending_proposal(alice_provider.storage(), &proposal_ref)
    .expect("The proposal was not found");
```

Commit messages

Commit messages are returned as `StagedCommit` objects. The proposals they cover can be inspected through different functions, depending on the proposal type. After the application has inspected the `StagedCommit` and approved all the proposals it covers, the `StagedCommit` can be merged in the current group state by calling the `.merge_staged_commit()` function. For more details, see the `StagedCommit` documentation.

```
if let ProcessedMessageContent::StagedCommitMessage(staged_commit) =
    alice_processed_message.into_content()
{
    // We expect a remove proposal
    let remove = staged_commit
        .remove_proposals()
        .next()
        .expect("Expected a proposal.");
    // Check that Bob was removed
    assert_eq!(
        remove.remove_proposal().removed(),
        bob_group.own_leaf_index()
    );
    // Check that Charlie removed Bob
    assert!(matches!(
        remove.sender(),
        Sender::Member(member) if *member == charlies_leaf_index
    ));
    // Merge staged commit
    alice_group
        .merge_staged_commit(alice_provider, *staged_commit)
        .expect("Error merging staged commit.");
}
```

Interpreting remove operations

Remove operations can have different meanings, such as:

- We left the group (by our own wish)
- We were removed from the group (by another member or a pre-configured sender)
- We removed another member from the group
- Another member left the group (by their own wish)
- Another member was removed from the group (by a member or a pre-configured sender, but not by us)

Since all remove operations only appear as a `QueuedRemoveProposal`, the `RemoveOperation` enum can be constructed from the remove proposal and the current group state to reflect the scenarios listed above.

```
    if let ProcessedMessageContent::StagedCommitMessage(staged_commit) =
        bob_processed_message.into_content()
    {
        let remove_proposal = staged_commit
            .remove_proposals()
            .next()
            .expect("An unexpected error occurred.");

        // We construct a RemoveOperation enum to help us interpret the remove
operation
        let remove_operation = RemoveOperation::new(remove_proposal, &bob_group)
            .expect("An unexpected Error occurred.");

        match remove_operation {
            RemoveOperation::WeLeft => unreachable!(),
            // We expect this variant, since Bob was removed by Charlie
            RemoveOperation::WeWereRemovedBy(member) => {
                assert!(matches!(member, Sender::Member(member) if member ==
charlies_leaf_index));
            }
            RemoveOperation::TheyLeft(_) => unreachable!(),
            RemoveOperation::TheyWereRemovedBy(_) => unreachable!(),
            RemoveOperation::WeRemovedThem(_) => unreachable!(),
        }

        // Merge staged Commit
        bob_group
            .merge_staged_commit(bob_provider, *staged_commit)
            .expect("Error merging staged commit.");
    } else {
        unreachable!("Expected a StagedCommit.");
    }
```

Persistence of Group Data

The state of a given `MlsGroup` instance is continuously written to the configured `StorageProvider`. Later, the `MlsGroup` can be loaded from the provider using the `load` constructor, which can be called with the respective storage provider as well as the `GroupId` of the group to be loaded. For this to work, the group must have been written to the provider previously.

Forward-Secrecy Considerations

OpenMLS uses the `StorageProvider` to store sensitive key material. To achieve forward-secrecy (i.e. to prevent an adversary from decrypting messages sent in the past if a client is compromised), OpenMLS frequently deletes previously used key material through calls to the `StorageProvider`. `StorageProvider` implementations must thus take care to ensure that values deleted through any of the `delete_` functions of the trait are irrevocably deleted and that no copies are kept.

Discarding commits

The delivery service may reject a commit sent by a client. In this case, the application needs to ensure that the local state remains the same as it was before the commit was staged.

Cleaning up local state after discarded commits

Generally, if a commit is discarded (e.g. due to being rejected by the Delivery Service), it can be cleaned up by the application in the following way:

```
// clear pending commit and reset state
alice_group
    .clear_pending_commit(alice_provider.storage())
    .unwrap();
```

In general, the application only needs to complete the cleanup above in order to fully restore the local state to the way it was before the commit was staged.

In several other cases, additional cleanup may need to be done.

ExternalJoin

If a staged commit containing an external join proposal must be discarded, the entire `MlsGroup` instance should be discarded by the application.

```
// delete the `MlsGroup`
bob_group
    .delete(bob_provider.storage())
    .expect("Could not delete the group");
```

PreSharedKey

In addition to clearing the staged commit, the application may also clear the pre-shared key from storage.

```
// clear the psk that was stored earlier, if necessary
alice_provider
    .storage()
    .delete_psk(&psk)
    .expect("Could not delete stored psk");

// clear pending commit and reset state
alice_group
    .clear_pending_commit(alice_provider.storage())
    .expect("Could not clear pending commit");
```

Self Update

The storage provider may also be used by the application to store signature keypairs. For self updates that update a signature keypair for the client, if the application has stored a new keypair in the storage provider at this point, it can be deleted from the storage provider here.

Credential validation

Credential validation is a process that allows a member to verify the validity of the credentials of other members in the group. The process is described in detail in the MLS protocol specification.

In practice, the application should check the validity of the credentials of other members in two instances:

- When joining a new group (by looking at the ratchet tree)
- When processing messages (by looking at a add & update proposals of a StagedCommit)

WebAssembly

OpenMLS can be built for WebAssembly. However, it does require two features that WebAssembly itself does not provide: access to secure randomness and the current time. Currently, this means that it can only run in a runtime that provides common JavaScript APIs (e.g. in the browser or node.js), accessed through the `web_sys` crate. You can enable the `js` feature on the `openmls` crate to signal that the APIs are available.

Fork Resolution

If members of a group merge different commits, the group state is called forked. At this point, the group members have different keys and will not be able to decrypt each others' messages. While this should not happen in normal operation, it may still occur due to bugs. When enabling the `fork-resolution-helpers` feature, OpenMLS comes with helpers to get a working group again. There are two helpers, and they use different mechanisms.

The `readd` helper removes and then re-adds members that are forked. This requires that the caller knows the set of members that are forked. It is relatively efficient, especially if only a small number of members forked.

The `reboot` helper creates a new group and helps with migrating the entire group state over. This includes extensions in the group context, as well as re-inviting all the members.

We provide examples for how to use both, and in the end provide some guidance on detecting forks.

`readd` Example

First, let's create a forked group. In this example, Alice creates a group and adds Bob. Then, they both merge different commits to add Charlie.

```
    // Alice creates a group
    let mut alice_group = MlsGroup::new(
        alice_provider,
        &alice_signature_keys,
        &mls_group_create_config,
        alice_credential.clone(),
    )
    .unwrap();

    // Alice adds Bob and merges the commit
    let add_bob_messages = alice_group
        .commit_builder()
        .propose_adds(vec![bob_kpb.key_package().clone()])
        .load_psks(alice_provider.storage())
        .unwrap()
        .build(
            alice_provider.rand(),
            alice_provider.crypto(),
            &alice_signature_keys,
            |_| true,
        )
        .unwrap()
        .stage_commit(alice_provider)
        .unwrap();

    alice_group.merge_pending_commit(alice_provider).unwrap();

    // Bob joins from the welcome
    let welcome = add_bob_messages.into_welcome().unwrap();
    let mut bob_group =
        StagedWelcome::new_from_welcome(bob_provider, mls_group_config,
welcome.clone(), None)
            .unwrap()
            .into_group(bob_provider)
            .unwrap();

    // Now Alice and Bob both add Charlie and merge their own commit.
    // This forks the group.
    let charlie_kpb = generate_key_package(
```

```
    ciphersuite,
    charlie_credential,
    Extensions::empty(),
    charlie_provider,
    &charlie_signature_keys,
);

let add_charlie_messages = alice_group
    .commit_builder()
    .propose_adds(vec![charlie_kpb.key_package().clone()])
    .load_psks(alice_provider.storage())
    .unwrap()
    .build(
        alice_provider.rand(),
        alice_provider.crypto(),
        &alice_signature_keys,
        |_| true,
    )
    .unwrap()
    .stage_commit(alice_provider)
    .unwrap();

bob_group
    .commit_builder()
    .propose_adds(vec![charlie_kpb.key_package().clone()])
    .load_psks(bob_provider.storage())
    .unwrap()
    .build(
        bob_provider.rand(),
        bob_provider.crypto(),
        &bob_signature_keys,
        |_| true,
    )
    .unwrap()
    .stage_commit(bob_provider)
    .unwrap();

alice_group.merge_pending_commit(alice_provider).unwrap();
bob_group.merge_pending_commit(bob_provider).unwrap();
```

```
    // Charlie joins using Alice's invite
    let welcome = add_charlie_messages.into_welcome().unwrap();
    let mut charlie_group =
        StagedWelcome::new_from_welcome(charlie_provider, mls_group_config, welcome,
None)
            .unwrap()
            .into_group(charlie_provider)
            .unwrap();

    // We should be forked now, double-check
    // Alice and Charlie are on the same state
    assert_eq!(
        alice_group.confirmation_tag(),
        charlie_group.confirmation_tag()
    );
    // But Bob is different from the other two
    assert_ne!(bob_group.confirmation_tag(), alice_group.confirmation_tag());
    assert_ne!(
        bob_group.confirmation_tag(),
        charlie_group.confirmation_tag()
    );
```

Then, Alice removes and re-adds Bob using the helper. We assume here that Alice knows that only Bob merged the wrong commit. This information needs to be transferred somehow, see Fork Detection. Notice how Alice needs to provide a new key package for Bob.

```
    // Let Alice re-add the members of the other partition (i.e. Bob)
    let bob_new_kpb = generate_key_package(
        ciphersuite,
        bob_credential,
        Extensions::empty(),
        bob_provider,
        &bob_signature_keys,
    );

    // Alice and Charlie are in the same partition
    let our_partition = &[alice_group.own_leaf_index(),
charlie_group.own_leaf_index()];
    let builder = alice_group.recover_fork_by_readding(our_partition).unwrap();

    // Here we iterate over the members of the complement partition to get their key
packages.
    // In this example this is trivial, but the pattern extends to more realistic
scenarios.
    let readded_key_packages = builder
        .complement_partition()
        .iter()
        .map(|member| {
            let basic_credential =
BasicCredential::try_from(member.credential.clone()).unwrap();
            match basic_credential.identity() {
                b"Bob" => bob_new_kpb.key_package().clone(),
                other => panic!(
                    "we only expect bob to be re-added, but found {:?}",
                    String::from_utf8(other.to_vec()).unwrap()
                ),
            }
        })
        .collect();

    // Specify the key packages to be re-added and create the commit
    let readd_messages = builder
        .provide_key_packages(readded_key_packages)
        .load_psks(alice_provider.storage())
        .unwrap()
```

```
            .build(
                alice_provider.rand(),
                alice_provider.crypto(),
                &alice_signature_keys,
                |_| true,
            )
            .unwrap()
            .stage_commit(alice_provider)
            .unwrap();

    // Make Bob re-join the group and Alice and Charlie merge the commit that adds
Bob.
    let (commit, welcome, _) = readd_messages.into_contents();
    let welcome = welcome.unwrap();
    let bob_group = StagedWelcome::new_from_welcome(bob_provider, mls_group_config,
welcome, None)
            .unwrap()
            .into_group(bob_provider)
            .unwrap();

    alice_group.merge_pending_commit(alice_provider).unwrap();

    if let ProcessedMessageContent::StagedCommitMessage(staged_commit) = charlie_group
            .process_message(charlie_provider, commit.into_protocol_message().unwrap())
            .unwrap()
            .into_content()
    {
        charlie_group
                .merge_staged_commit(charlie_provider, *staged_commit)
                .unwrap()
    } else {
        panic!("expected a commit")
    }

    // The fork should be fixed now, double-check
    assert_eq!(alice_group.confirmation_tag(), bob_group.confirmation_tag());
    assert_eq!(
        alice_group.confirmation_tag(),
        charlie_group.confirmation_tag()
    );
```

```
    assert_eq!(
        charlie_group.confirmation_tag(),
        bob_group.confirmation_tag()
    );
```

In the end, they all can communicate again.

`reboot` Example

Again, let's create a forked group. In this example, Alice creates a group and adds Bob. Then, they both merge different commits to add Charlie.

```
// Alice creates a group
let mut alice_group = MlsGroup::new(
    alice_provider,
    &alice_signature_keys,
    &mls_group_create_config,
    alice_credential.clone(),
)
.unwrap();

// Alice adds Bob and merges the commit
let add_bob_messages = alice_group
    .commit_builder()
    .propose_adds(vec![bob_kpb.key_package().clone()])
    .load_psks(alice_provider.storage())
    .unwrap()
    .build(
        alice_provider.rand(),
        alice_provider.crypto(),
        &alice_signature_keys,
        |_| true,
    )
    .unwrap()
    .stage_commit(alice_provider)
    .unwrap();

alice_group.merge_pending_commit(alice_provider).unwrap();

// Bob joins from the welcome
let welcome = add_bob_messages.into_welcome().unwrap();
let mut bob_group =
    StagedWelcome::new_from_welcome(bob_provider, mls_group_config, welcome, None)
        .unwrap()
        .into_group(bob_provider)
        .unwrap();

// Now Alice and Bob both add Charlie and merge their own commit.
// This forks the group.
let charlie_kpb = generate_key_package(
    ciphersuite,
```

```
    charlie_credential.clone(),
    Extensions::empty(),
    charlie_provider,
    &charlie_signature_keys,
);

let add_charlie_messages = alice_group
    .commit_builder()
    .propose_adds(vec![charlie_kpb.key_package().clone()])
    .load_psks(alice_provider.storage())
    .unwrap()
    .build(
        alice_provider.rand(),
        alice_provider.crypto(),
        &alice_signature_keys,
        |_| true,
    )
    .unwrap()
    .stage_commit(alice_provider)
    .unwrap();

bob_group
    .commit_builder()
    .propose_adds(vec![charlie_kpb.key_package().clone()])
    .load_psks(bob_provider.storage())
    .unwrap()
    .build(
        bob_provider.rand(),
        bob_provider.crypto(),
        &bob_signature_keys,
        |_| true,
    )
    .unwrap()
    .stage_commit(bob_provider)
    .unwrap();

alice_group.merge_pending_commit(alice_provider).unwrap();
bob_group.merge_pending_commit(bob_provider).unwrap();

// Charlie joins using Alice's invite
```

```
    let welcome = add_charlie_messages.into_welcome().unwrap();
    let charlie_group =
        StagedWelcome::new_from_welcome(charlie_provider, mls_group_config, welcome,
None)
            .unwrap()
            .into_group(charlie_provider)
            .unwrap();

    // We shoulkd be forked now, double-check
    // Alice and Charlie are on the same state
    assert_eq!(
        alice_group.confirmation_tag(),
        charlie_group.confirmation_tag()
    );
    // But Bob is different from the other two
    assert_ne!(bob_group.confirmation_tag(), alice_group.confirmation_tag());
    assert_ne!(
        bob_group.confirmation_tag(),
        charlie_group.confirmation_tag()
    );
```

Then, Alice sets up a new group and adds everyone from the old group. In this approach, she not only needs to provide key packages for all members, but also set a new group id and migrate the group context extensions, because these might be contain e.g. the old group id. This is the responsibility of the application, so the API just exposes the old extensions and expects the new ones.

```
    // Let Alice reboot the group. For that she needs new key packages for Bob and
Charlie, a;s
    // well as a new group ID.
    let bob_new_kpb = generate_key_package(
        ciphersuite,
        bob_credential,
        Extensions::empty(),
        bob_provider,
        &bob_signature_keys,
    );

    let charlie_new_kpb = generate_key_package(
        ciphersuite,
        charlie_credential,
        Extensions::empty(),
        charlie_provider,
        &charlie_signature_keys,
    );

    let new_group_id: GroupId = GroupId::from_slice(
        alice_group
            .group_id()
            .as_slice()
            .iter()
            .copied()
            .chain(b"-new".iter().copied())
            .collect::<Vec<_>>()
            .as_slice(),
    );

    let (mut alice_group, reboot_messages) = alice_group
        .reboot(new_group_id)
        .finish(
            Extensions::empty(),
            vec![
                bob_new_kpb.key_package().clone(),
                charlie_new_kpb.key_package().clone(),
            ],
            // We can use this closure to add more proposals to the commit builder
```

```
that is used to
            // create the commit that readds all the other members, but in this case
we will leave
            // it as-is.
            |builder| builder,
            alice_provider,
            &alice_signature_keys,
            alice_credential,
        )
        .unwrap();

    alice_group.merge_pending_commit(alice_provider).unwrap();

    // Bob and Charlie join the new group
    let welcome = reboot_messages.into_welcome().unwrap();
    let bob_group =
        StagedWelcome::new_from_welcome(bob_provider, mls_group_config,
welcome.clone(), None)
            .unwrap()
            .into_group(bob_provider)
            .unwrap();
    assert_eq!(bob_group.own_leaf_index(), LeafNodeIndex::new(1));

    let charlie_group =
        StagedWelcome::new_from_welcome(charlie_provider, mls_group_config, welcome,
None)
            .unwrap()
            .into_group(charlie_provider)
            .unwrap();
    assert_eq!(charlie_group.own_leaf_index(), LeafNodeIndex::new(2));

    // The fork should be fixed now, double-check
    assert_eq!(alice_group.confirmation_tag(), bob_group.confirmation_tag());
    assert_eq!(
        alice_group.confirmation_tag(),
        charlie_group.confirmation_tag()
    );
    assert_eq!(
        bob_group.confirmation_tag(),
        charlie_group.confirmation_tag()
```

```
);
```

In the end, they all can communicate again.

Fork Detection

Before initiating fork resolution, we first need to detect that a fork happened. In addition, for using the `readd` mechanism, we also need to know the members that forked.

One simple technique that may work, depending on how the delivery service works, is to consider all incoming non-decryptable messages as a sign that there is a fork. However, this may lead to false positives and is not enough to know the membership.

One way to learn about this that every member send a message when they merges a commit, encrypted for the old epoch, that contains the hash of the commit they are merging. This way, all group members know which commits are merged, and the `readd` strategy can be used to resolve possible forks.

Traits & External Types

OpenMLS defines several traits that have to be implemented to use OpenMLS. The main goal is to allow OpenMLS to use different implementations for its cryptographic primitives, persistence, and random number generation. This should make it possible to plug in anything from WebCrypto to secure enclaves.

Using storage

The store is probably one of the most interesting traits because applications that use OpenMLS will interact with it. See the StorageProvider trait description for details.

In the following examples, we have a `ciphersuite` and a `provider` (`OpenMlsProvider`).

```
// First we generate a credential and key package for our user.
let credential = BasicCredential::new(b"User ID".to_vec());
let signature_keys = SignatureKeyPair::new(ciphersuite.into()).unwrap();

// This key package includes the private init and encryption key as well.
// See [`KeyPackageBundle`].
let key_package = KeyPackage::builder()
    .build(
        ciphersuite,
        provider,
        &signature_keys,
        CredentialWithKey {
            credential: credential.into(),
            signature_key: signature_keys.to_public_vec().into(),
        },
    )
    .unwrap();
```

Retrieving a value from the store is as simple as calling `read`. The retrieved key package bundles the private keys for the init and encryption keys as well.

```
// Read the key package
let read_key_package: Option<KeyPackageBundle> = provider
    .storage()
    .key_package(&hash_ref)
    .expect("Error reading key package");
assert_eq!(
    read_key_package.unwrap().key_package(),
    key_package.key_package()
);
```

The `delete` is called with the identifier to delete a value.

```
// Delete the key package
let hash_ref = key_package
    .key_package()
    .hash_ref(provider.crypto())
    .unwrap();
provider
    .storage()
    .delete_key_package(&hash_ref)
    .expect("Error deleting key package");
```

OpenMLS Traits

> ⚠️ **These traits are responsible for all cryptographic operations and randomness within OpenMLS. Please ensure you know what you're doing when implementing your own versions.**

Because implementing the `OpenMLSCryptoProvider` is challenging, requires tremendous care, and is not what the average OpenMLS consumer wants to (or should) do, we provide two implementations that can be used.

- Rust Crypto
- Libcrux Crypto

Rust Crypto Provider The go-to default at the moment is an implementation using commonly used, native Rust crypto implementations.

Libcrux Crypto Provider A crypto provider backed by the high-assurance cryptography library [libcrux]. Currently only supports relatively modern x86 and amd64 CPUs, as it requires AES-NI, SIMD and AVX.

The Traits

There are 4 different traits defined in the OpenMLS traits crate.

OpenMlsRand

This trait defines two functions to generate arrays and vectors, and is used by OpenMLS to generate randomness for key generation and random identifiers. While there is the commonly used rand crate, not all implementations use it. OpenMLS, therefore, defines its own randomness trait that needs to be implemented by an OpenMLS crypto provider. It simply needs to implement two functions to generate cryptographically secure randomness and store it in an array or vector.

```
pub trait OpenMlsRand {
    type Error: std::error::Error + Debug;

    /// Fill an array with random bytes.
    fn random_array<const N: usize>(&self) -> Result<[u8; N], Self::Error>;

    /// Fill a vector of length `len` with bytes.
    fn random_vec(&self, len: usize) -> Result<Vec<u8>, Self::Error>;
}
```

OpenMlsCrypto

This trait defines all cryptographic functions required by OpenMLS. In particular:

- HKDF
- Hashing
- AEAD
- Signatures
- HPKE

StorageProvider

This trait defines an API for a storage backend that is used for all OpenMLS persistence.

The store provides functions for reading and updating stored values. Each sort of value has separate

methods for accessing or mutating the state. In order to decouple the provider from the OpenMLS implementation, while still having legible types at the provider, there are traits that mirror all the types stored by OpenMLS. The provider methods use values constrained by these traits as as arguments.

```rust
/// Each trait in this module corresponds to a type. Some are used as keys, some as
/// entities, and some both. Therefore, the Key and/or Entity traits also need to be
implemented.
pub mod traits {
    use super::{Entity, Key};

    // traits for keys, one per data type
    pub trait GroupId<const VERSION: u16>: Key<VERSION> {}
    pub trait SignaturePublicKey<const VERSION: u16>: Key<VERSION> {}
    pub trait HashReference<const VERSION: u16>: Key<VERSION> {}
    pub trait PskId<const VERSION: u16>: Key<VERSION> {}
    pub trait EncryptionKey<const VERSION: u16>: Key<VERSION> {}
    pub trait EpochKey<const VERSION: u16>: Key<VERSION> {}

    // traits for entity, one per type
    pub trait QueuedProposal<const VERSION: u16>: Entity<VERSION> {}
    pub trait TreeSync<const VERSION: u16>: Entity<VERSION> {}
    pub trait GroupContext<const VERSION: u16>: Entity<VERSION> {}
    pub trait InterimTranscriptHash<const VERSION: u16>: Entity<VERSION> {}
    pub trait ConfirmationTag<const VERSION: u16>: Entity<VERSION> {}
    pub trait SignatureKeyPair<const VERSION: u16>: Entity<VERSION> {}
    pub trait PskBundle<const VERSION: u16>: Entity<VERSION> {}
    pub trait HpkeKeyPair<const VERSION: u16>: Entity<VERSION> {}
    pub trait GroupState<const VERSION: u16>: Entity<VERSION> {}
    pub trait GroupEpochSecrets<const VERSION: u16>: Entity<VERSION> {}
    pub trait LeafNodeIndex<const VERSION: u16>: Entity<VERSION> {}
    pub trait MessageSecrets<const VERSION: u16>: Entity<VERSION> {}
    pub trait ResumptionPskStore<const VERSION: u16>: Entity<VERSION> {}
    pub trait KeyPackage<const VERSION: u16>: Entity<VERSION> {}
    pub trait MlsGroupJoinConfig<const VERSION: u16>: Entity<VERSION> {}
    pub trait LeafNode<const VERSION: u16>: Entity<VERSION> {}
    pub trait ApplicationExportTree<const VERSION: u16>: Entity<VERSION> {}

    // traits for types that implement both
    pub trait ProposalRef<const VERSION: u16>: Entity<VERSION> + Key<VERSION> {}
}
```

The traits are generic over a `VERSION`, which is used to ensure that the values that are persisted can

be upgraded when OpenMLS changes the stored structs.

The traits used as arguments to the storage methods are constrained to implement the `Key` or `Entity` traits as well, depending on whether they are only used for addressing (in which case they are a `Key`) or whether they represent a stored value (in which case they are an `Entity`).

```
/// Key is a trait implemented by all types that serve as a key (in the database
sense) to in the
/// storage. For example, a GroupId is a key to the stored entities for the group with
that id.
/// The point of a key is not to be stored, it's to address something that is stored.
pub trait Key<const VERSION: u16>: Serialize {}


/// Entity is a trait implemented by the values being stored.
pub trait Entity<const VERSION: u16>: Serialize + DeserializeOwned {}
```

An implementation of the storage trait should ensure that it can address and efficiently handle values.

Example: Key packages

This is only an example, but it illustrates that the application may need to do more when it comes to implementing storage.

Key packages are only deleted by OpenMLS when they are used and *not* last resort key packages (which may be used multiple times). The application needs to implement some logic to manage last resort key packages.

```
    fn write_key_package<
        HashReference: traits::HashReference<VERSION>,
        KeyPackage: traits::KeyPackage<VERSION>,
    >(
        &self,
        hash_ref: &HashReference,
        key_package: &KeyPackage,
    ) -> Result<(), Self::Error>;
```

The application may store the hash references in a separate list with a validity period.

```
fn write_key_package<
    HashReference: traits::HashReference<VERSION>,
    KeyPackage: traits::KeyPackage<VERSION>,
>(
    &self,
    hash_ref: &HashReference,
    key_package: &KeyPackage,
) -> Result<(), Self::Error> {
    // Get the validity from the application in some way.
    let validity = self.get_validity(hash_ref);

    // Store the reference and its validity period.
    self.store_hash_ref(hash_ref, validity);

    // Store the actual key package.
    self.store_key_package(hash_ref, key_package);
}
```

This allows the application to iterate over the hash references and delete outdated key packages.

OpenMlsProvider

Additionally, there's a wrapper trait defined that is expected to be passed into the public OpenMLS

API. Some OpenMLS APIs require only one of the sub-traits, though.

```
pub trait OpenMlsProvider {
    type CryptoProvider: crypto::OpenMlsCrypto;
    type RandProvider: random::OpenMlsRand;
    type StorageProvider: storage::StorageProvider<{ storage::CURRENT_VERSION }>;

    // Get the storage provider.
    fn storage(&self) -> &Self::StorageProvider;

    /// Get the crypto provider.
    fn crypto(&self) -> &Self::CryptoProvider;

    /// Get the randomness provider.
    fn rand(&self) -> &Self::RandProvider;
}
```

Implementation Notes

It is not necessary to implement all sub-traits if one functionality is missing. Suppose you want to use a persisting storage provider. In that case, it is sufficient to do a new implementation of the `StorageProvider` trait and combine it with one of the provided crypto and randomness trait implementations.

External Types

For interoperability, this crate also defines several types and algorithm identifiers.

AEADs

The following AEADs are defined.

```
#[derive(Debug, PartialEq, Eq, Clone, Copy, Serialize, Deserialize)]
#[repr(u16)]
/// AEAD types
pub enum AeadType {
    /// AES GCM 128
    Aes128Gcm = 0x0001,

    /// AES GCM 256
    Aes256Gcm = 0x0002,
```

An AEAD provides the following functions to get the according values for each algorithm.

- `tag_size`
- `key_size`
- `nonce_size`

Hashing

The following hash algorithms are defined.

```
#[repr(u8)]
#[allow(non_camel_case_types)]
/// Hash types
pub enum HashType {
    Sha2_256 = 0x04,
```

A hash algorithm provides the following functions to get the according values for each algorithm.

- `size`

Signatures

The following signature schemes are defined.

```
    TlsDeserializeBytes,
    TlsSize,
)]
#[repr(u16)]
pub enum SignatureScheme {
    /// ECDSA_SECP256R1_SHA256
    ECDSA_SECP256R1_SHA256 = 0x0403,
    /// ECDSA_SECP384R1_SHA384
    ECDSA_SECP384R1_SHA384 = 0x0503,
    /// ECDSA_SECP521R1_SHA512
    ECDSA_SECP521R1_SHA512 = 0x0603,
    /// ED25519
```

HPKE Types

The HPKE implementation is part of the crypto provider as well. The crate, therefore, defines the necessary types too.

The HPKE algorithms are defined as follows.

```rust
/// KEM Types for HPKE
#[derive(PartialEq, Eq, Copy, Clone, Debug, Serialize, Deserialize)]
#[repr(u16)]
pub enum HpkeKemType {
    /// DH KEM on P256
    DhKemP256 = 0x0010,

    /// DH KEM on P384
    DhKemP384 = 0x0011,

    /// DH KEM on P521
    DhKemP521 = 0x0012,

    /// DH KEM on x25519
    DhKem25519 = 0x0020,


    /// XWing combiner for ML-KEM and X25519
    XWingKemDraft6 = 0x004D,
}

/// KDF Types for HPKE
#[derive(PartialEq, Eq, Copy, Clone, Debug, Serialize, Deserialize)]
#[repr(u16)]
pub enum HpkeKdfType {
    /// HKDF SHA 256
    HkdfSha256 = 0x0001,
```

```
    /// HKDF SHA 512
    HkdfSha512 = 0x0003,
}

/// AEAD Types for HPKE.
#[derive(Clone, Copy, Debug, PartialEq, Eq, Serialize, Deserialize)]
#[repr(u16)]
pub enum HpkeAeadType {
    /// AES GCM 128
    AesGcm128 = 0x0001,

    /// AES GCM 256
    AesGcm256 = 0x0002,
```

In addition, helper structs for `HpkeCiphertext` and `HpkeKeyPair` are defined.

```
///     opaque kem_output<V>;
///     opaque ciphertext<V>;
/// } HPKECiphertext;
/// ```


    Eq,
    Clone,
    Serialize,
    Deserialize,
```

Message Validation

OpenMLS implements a variety of syntactical and semantical checks, both when parsing and processing incoming commits and when creating own commits.

Validation steps

Validation is enforced using Rust's type system. The chain of functions used to process incoming messages is described in the chapter on Processing incoming messages, where each function takes a distinct type as input and produces a distinct type as output, thus ensuring that the individual steps can't be skipped. We now detail which step performs which validation checks.

Syntax validation

Incoming messages in the shape of a byte string can only be deserialized into a `MlsMessageIn` struct. Deserialization ensures that the message is a syntactically correct MLS message, i.e., either a PublicMessage or a PrivateMessage. Further syntax checks are applied for the latter case once the message is decrypted.

Semantic validation

Every function in the processing chain performs several semantic validation steps. For a list of these steps, see below. In the following, we will give a brief overview of which function performs which category of checks.

Wire format policy and basic message consistency validation

`MlsMessageIn` struct instances can be passed into the `.parse_message()` function of the `MlsGroup` API, which validates that the message conforms to the group's wire format policy. The function also performs several basic semantic validation steps, such as consistency of Group id, Epoch, and Sender data between message and group (`ValSem002` - `ValSem007`). It also checks if the sender type (e.g., `Member` , `NewMember` , etc.) matches the type of the message (`ValSem112`), as well as the presence of a path in case of an External Commit (`ValSem246`).

`.parse_message()` then returns an `UnverifiedMessage` struct instance, which can in turn be used as input for `.process_unverified_message()` .

Message-specific semantic validation

`.process_unverified_message()` performs all other semantic validation steps. In particular, it ensures that ...

- the message is correctly authenticated by a signature (`ValSem010`), membership tag (`ValSem008`), and confirmation tag (`ValSem205`),
- proposals are valid relative to one another and the current group state, e.g., no redundant adds or removes targeting non-members (`ValSem101` - `ValSem112`),
- commits are valid relative to the group state and the proposals it covers (`ValSem200` - `ValSem205`) and
- external commits are valid according to the spec (`ValSem240` - `ValSem245` , `ValSem247` is checked as part of `ValSem010`).

After performing these steps, messages are returned as `ProcessedMessage` s that the application can either use immediately (application messages) or inspect and decide if they find them valid according to the application's policy (proposals and commits). Proposals can then be stored in the proposal queue via `.store_pending_proposal()` , while commits can be merged into the group state via `.merge_staged_commit()` .

Detailed list of validation steps

The following is a list of the individual semantic validation steps performed by OpenMLS, including the location of the tests.

Semantic validation of message framing

ValidationStep	Description	Implemented	Tested	Test
ValSem002	Group id	✅	✅	openmls/src/g test_framing_v
ValSem003	Epoch	✅	✅	openmls/src/g test_framing_v
ValSem004	Sender: Member: check the sender points to a non-blank leaf	✅	✅	openmls/src/g test_framing_v
ValSem005	Application messages must use ciphertext	✅	✅	openmls/src/g test_framing_v
ValSem006	Ciphertext: decryption needs to work	✅	✅	openmls/src/g test_framing_v
ValSem007	Membership tag presence	✅	✅	openmls/src/g test_framing_v
ValSem008	Membership tag verification	✅	✅	openmls/src/g test_framing_v
ValSem009	Confirmation tag presence	✅	✅	openmls/src/g test_framing_v
ValSem010	Signature verification	✅	✅	openmls/src/g test_framing_v

ValidationStep	Description	Implemented	Tested	Test
`ValSem011`	PrivateMessageContent padding must be all-zero	✅	✅	`openmls/src/g test_framing.r`

Semantic validation of proposals covered by a Commit

ValidationStep	Description	Implemented	Tested	Test File
`ValSem101`	Add Proposal: Signature public key in proposals must be unique among proposals & members	✅	✅	`openmls/src/group/tests, test_proposal_validation`
`ValSem102`	Add Proposal: Init key in proposals must be unique among proposals	✅	✅	`openmls/src/group/tests, test_proposal_validation`

ValidationStep	Description	Implemented	Tested	Test File
`ValSem103`	Add Proposal: Encryption key in proposals must be unique among proposals & members	✅	✅	`openmls/src/group/tests/test_proposal_validation`
`ValSem104`	Add Proposal: Init key and encryption key must be different	✅	✅	`openmls/src/group/tests/test_proposal_validation`
`ValSem105`	Add Proposal: Ciphersuite & protocol version must match the group	✅	✅	`openmls/src/group/tests/test_proposal_validation`
`ValSem106`	Add Proposal: required capabilities	✅	✅	`openmls/src/group/tests/test_proposal_validation`
`ValSem107`	Remove Proposal:	✅	✅	`openmls/src/group/tests/test_proposal_validation`

ValidationStep	Description	Implemented	Tested	Test File
	Removed member must be unique among proposals			
`ValSem108`	Remove Proposal: Removed member must be an existing group member	✅	✅	`openmls/src/group/tests/test_proposal_validation`
`ValSem109`	Update Proposal: required capabilities	✅	✅	`openmls/src/group/tests/test_proposal_validation`
`ValSem110`	Update Proposal: Encryption key must be unique among proposals & members	✅	✅	`openmls/src/group/tests/test_proposal_validation`
`ValSem111`	Update Proposal: The sender	✅	✅	`openmls/src/group/tests/test_proposal_validation`

ValidationStep	Description	Implemented	Tested	Test File
	of a full Commit must not include own update proposals			
`ValSem112`	Update Proposal: The sender of a standalone update proposal must be of type member	✅	✅	`openmls/src/group/tests/test_proposal_validation`
`ValSem113`	All Proposals: The proposal type must be supported by all members of the group	✅	✅	`openmls/src/group/tests/test_proposal_validation`

Commit message validation

ValidationStep	Description	Implemented	Tested	Test
`ValSem200`	Commit must not cover inline self Remove proposal	✅	✅	`openmls/src/ test_commit_v`
`ValSem201`	Path must be present, if at least one proposal requires a path	✅	✅	`openmls/src/ test_commit_v`
`ValSem202`	Path must be the right length	✅	✅	`openmls/src/ test_commit_v`
`ValSem203`	Path secrets must decrypt correctly	✅	✅	`openmls/src/ test_commit_v`
`ValSem204`	Public keys from Path must be verified and match the private keys from the direct path	✅	✅	`openmls/src/ test_commit_v`
`ValSem205`	Confirmation tag must be successfully verified	✅	✅	`openmls/src/ test_commit_v`
`ValSem206`	Path leaf node encryption key must be unique among proposals & members	✅	✅	`openmls/src/ test_commit_v`
`ValSem207`	Path encryption keys must be unique among proposals & members	✅	✅	`openmls/src/ test_commit_v`
`ValSem208`	Only one GroupContextExtensions proposal in a commit	✅		
`ValSem209`	GroupContextExtensions	✅		

ValidationStep	Description	Implemented	Tested	Test
	proposals may only contain extensions support by all members			

External Commit message validation

ValidationStep	Description	Implemented	Tested	Test File
`ValSem240`	External Commit must cover at least one inline ExternalInit proposal	✅	✅	`openmls/src/group/tes test_external_commit_v`
`ValSem241`	External Commit must cover at most one inline ExternalInit proposal	✅	✅	`openmls/src/group/tes test_external_commit_v`
`ValSem242`	External Commit must only cover inline proposal in allowlist (ExternalInit, Remove, PreSharedKey)	✅	✅	`openmls/src/group/tes test_external_commit_v`

ValidationStep	Description	Implemented	Tested	Test File
`ValSem244`	External Commit must not include any proposals by reference	✅	✅	`openmls/src/group/tes test_external_commit_v`
`ValSem245`	External Commit must contain a path	✅	✅	`openmls/src/group/tes test_external_commit_v`
`ValSem246`	External Commit signature must be verified using the credential in the path KeyPackage	✅	✅	`openmls/src/group/tes test_external_commit_v`

Ratchet tree validation

ValidationStep	Description	Implemented	Tested	Test File
`ValSem300`	Exported ratchet trees must not have trailing blank nodes.	Yes	Yes	`openmls/src/treesync/mod.rs`

PSK Validation

ValidationStep	Description	Implemented	Tested	Test File
`ValSem400`	The application SHOULD specify an upper limit on the number of past epochs for which the resumption_psk may be stored.	❌	❌	https://github.com/op openmls/issues/1122
`ValSem401`	The nonce of a PreSharedKeyID must have length KDF.Nh.	✅	✅	`openmls/src/group/t test_proposal_valida`
`ValSem402`	PSK in proposal must be of type Resumption (with usage Application) or External.	✅	✅	`openmls/src/group/t test_proposal_valida`
`ValSem403`	Proposal list must not contain multiple PreSharedKey proposals that reference the same PreSharedKeyID.	✅	❌	https://github.com/op openmls/issues/1335

App Validation

NOTE: This chapter described the validation steps an application, using OpenMLS, has to perform for safe operation of the MLS protocol.

⚠️ This chapter is work in progress (see #1504).

Credential Validation

Acceptable Presented Identifiers

The application using MLS is responsible for specifying which identifiers it finds acceptable for each member in a group. In other words, following the model that [RFC6125] describes for TLS, the application maintains a list of "reference identifiers" for the members of a group, and the credentials provide "presented identifiers". A member of a group is authenticated by first validating that the member's credential legitimately represents some presented identifiers, and then ensuring that the reference identifiers for the member are authenticated by those presented identifiers

-- RFC9420, Section 5.3.1

Validity of Updated Presented Identifiers

> In cases where a member's credential is being replaced, such as the Update and Commit cases above, the AS MUST also verify that the set of presented identifiers in the new credential is valid as a successor to the set of presented identifiers in the old credential, according to the application's policy.
>
> -- RFC9420, Section 5.3.1

Application ID is Not Authenticed by AS

> However, applications MUST NOT rely on the data in an application_id extension as if it were authenticated by the Authentication Service, and SHOULD gracefully handle cases where the identifier presented is not unique.
>
> -- RFC9420, Section 5.3.3

LeafNode Validation

Specifying the Maximum Total Acceptable Lifetime

> Applications MUST define a maximum total lifetime that is acceptable for a LeafNode, and reject any LeafNode where the total lifetime is longer than this duration. In order to avoid disagreements about whether a LeafNode has a valid lifetime, the clients in a group SHOULD maintain time synchronization (e.g., using the Network Time Protocol [RFC5905]).

-- RFC9420, Section 7.2

PrivateMessage Validation

Structure of AAD is Application-Defined

> It is up to the application to decide what authenticated_data to provide and how much padding to add to a given message (if any). The overall size of the AAD and ciphertext MUST fit within the limits established for the group's AEAD algorithm in [CFRG-AEAD-LIMITS].
>
> -- RFC9420, Section 6.3.1

Therefore, the application must also validate whether the AAD adheres to the prescribed format.

Proposal Validation

When processing a commit, the application has to ensure that the application specific semantic checks for the validity of the committed proposals are performed.

This should be done on the `StagedCommit`. Also see the Message Processing chapter

```
if let ProcessedMessageContent::StagedCommitMessage(staged_commit) =
    alice_processed_message.into_content()
{
    // We expect a remove proposal
    let remove = staged_commit
        .remove_proposals()
        .next()
        .expect("Expected a proposal.");
    // Check that Bob was removed
    assert_eq!(
        remove.remove_proposal().removed(),
        bob_group.own_leaf_index()
    );
    // Check that Charlie removed Bob
    assert!(matches!(
        remove.sender(),
        Sender::Member(member) if *member == charlies_leaf_index
    ));
    // Merge staged commit
    alice_group
        .merge_staged_commit(alice_provider, *staged_commit)
        .expect("Error merging staged commit.");
}
```

External Commits

The RFC requires the following check

> At most one Remove proposal, with which the joiner removes an old version of themselves. If a Remove proposal is present, then the LeafNode in the path field of the external Commit MUST meet the same criteria as would the LeafNode in an Update for the removed leaf (see Section 12.1.2). In particular, the credential in the LeafNode MUST present a set of identifiers that is acceptable to the application for the removed participant.

Since OpenMLS does not know the relevant policies, the application MUST ensure that the credentials are checked according to the policy.

Performance

How does OpenMLS (and MLS in general) perform in different settings?

Performance measurements are implemented here and can be run with `cargo bench --bench group`. Check which scenarios and group sizes are enabled in the code.

OpenMLS Performance Spreadsheet

Real World Scenarios

Stable group

Many private groups follow this model.

- Group is created by user P1
- P1 invites a set of N other users
- The group is used for messaging between the N+1 members
- Every X messages, one user in the group sends an update

Somewhat stable group

This can model a company or team-wide group where regularly but infrequently, users are added, and users leave.

- Group is created by user P1
- P1 invites a set of N other users

- The group is used for messaging between the members
- Every X messages, one user in the group sends an update
- Every Y messages, Q users are added
- Every Z messages, R users are removed

High fluctuation group

This models public groups where users frequently join and leave. Real-time scenarios such as gather.town are examples of high-fluctuation groups. It is the same scenario as the somewhat stable group but with a very small Y and Z.

Extreme Scenarios

In addition to the three scenarios above extreme and corner cases are interesting.

Every second leaf is blank

Only every second leaf in the tree is non-blank.

Use Case Scenarios

A collection of common use cases/flows from everyday scenarios.

Long-time offline device

Suppose a device has been offline for a while. In that case, it has to process a large number of application and protocol messages.

Tree scenarios

In addition to the scenarios above, it is interesting to look at the same scenario but with different states of the tree. For example, take the stable group with N members messaging each other. What is the performance difference between a message sent right after group setup, i.e., each member only joined the group without other messages being sent, and a tree where every member has sent an update before the message?

Measurements

- Group creation
  - create group
  - create proposals
  - create welcome
  - apply commit
- Join group
  - create group from welcome
- Send application message
- Receive application message
- Send update
  - create proposal
  - create commit
  - apply commit
- Receive update

    - apply commit
- Add user sender
    - create proposal
    - create welcome
    - apply commit
- Existing user getting an add
    - apply commit
- Remove user sender
    - create proposal
    - create commit
    - apply commit
- Existing user getting a remove
    - apply commit

Forward Secrecy

OpenMLS drops key material immediately after a given key is no longer required by the protocol to achieve forward secrecy. For some keys, this is simple, as they are used only once, and there is no need to store them for later use. However, for other keys, the time of deletion is a result of a trade-off between functionality and forward secrecy. For example, it can be desirable to keep the `SecretTree` of past epochs for a while to allow decryption of straggling application messages sent in previous epochs.

In this chapter, we detail how we achieve forward secrecy for the different types of keys used throughout MLS.

Ratchet Tree

The ratchet tree contains the secret key material of the client's leaf, as well (potentially) that of nodes in its direct path. The secrets in the tree are changed in the same way as the tree itself: via the merge of a previously prepared diff.

Commit Creation

Upon the creation of a commit, any fresh key material introduced by the committer is stored in the diff. It exists alongside the key material of the ratchet tree before the commit until the client merges the diff, upon which the key material in the original ratchet tree is dropped.

Because the client cannot know if the commit it creates will conflict with another commit created by another client for the same epoch, it MUST wait for the acknowledgement from the Delivery Service before merging the diff and dropping the previous ratchet tree.

Commit Processing

Upon receiving a commit from another group member, the client processes the commit until they have a `StagedCommit`, which in turn contains a ratchet tree diff. The diff contains any potential key material they decrypted from the commit and any potential key material that was introduced to the tree as part of an update that someone else committed for them. The key material in the original ratchet tree is dropped as soon as the `StagedCommit` (and thus the diff) is merged into the tree.

Sending application messages

When an application message is created, the corresponding encryption key is derived from the `SecretTree` and immediately discarded after encrypting the message to guarantee the best possible Forward Secrecy. This means that the message author cannot decrypt application messages. If access to the message's content is required after creating the message, a copy of the plaintext message should be kept by the application.

Receiving encrypted messages

When an encrypted message is received, the corresponding decryption key is derived from the `SecretTree`. By default, the key material is discarded immediately after decryption for the best possible Forward Secrecy. In some cases, the Delivery Service cannot guarantee reliable operation, and applications need to be more tolerant to accommodate this – at the expense of Forward Secrecy.

OpenMLS can address 3 scenarios:

- The Delivery Service cannot guarantee that application messages from one epoch are sent before the beginning of the next epoch. To address this, applications can configure their groups to keep the necessary key material around for past epochs by setting the

`max_past_epochs` field in the `MlsGroupCreateConfig` to the desired number of epochs.

- The Delivery Service cannot guarantee that application messages will arrive in order within the same epoch. To address this, applications can configure the `out_of_order_tolerance` parameter of the `SenderRatchetConfiguration`. The configuration can be set as the `sender_ratchet_configuration` parameter of the `MlsGroupCreateConfig`.
- The Delivery Service cannot guarantee that application messages won't be dropped within the same epoch. To address this, applications can configure the `maximum_forward_distance` parameter of the `SenderRatchetConfiguration`. The configuration can be set as the `sender_ratchet_configuration` parameter of the `MlsGroupCreateConfig`.

Release management

The process for releasing a new version of OpenMLS.

Versioning

The versioning follows the Rust and semantic versioning guidelines.

Release Notes

Release notes are published on GitHub with a full changelog and a discussion in the "Release" section. In addition, the release notes are prepended to the CHANGELOG file in each crate's root folder. The entries in the CHANGELOG file should follow the keep a changelog guide.

Pre-release strategy

Before releasing a minor or major version of the OpenMLS crate, a pre-release version must be published to crates.io. Pre-release versions are defined by appending a hyphen, and a series of dot-separated identifiers, i.e., `-rc.x` where `x` gets counted up starting at 1. Pre-releases must be tagged but don't require release notes or other documentation. It is also sufficient to tag only the most high-level crate being published.

Crates in this Repository

The crates must be published in the order below.

- Traits
- Memory Keystore
- Rust Crypto provider
- OpenMLS

Release note and changelog template

```
## 0.0.0 (2022-02-22)

### Added

- the feature ([#000])

### Changed

- the change ([#000])

### Deprecated

- the deprecated feature ([#000])

### Removed

- the removed feature ([#000])

### Fixed

- the fixed bug ([#000])

### Security

- the fixed security bug ([#000])

[#000]: https://github.com/openmls/openmls/pull/000
```

Release checklist

- ☐ If this is a minor or major release, has a pre-release version been published at least a week before the release?
  - If not, first do so and push the release one week.
- ☐ Describe the release in the CHANGELOG.md file of each crate.

- [ ] Create and publish a git tag for each crate, e.g. `openmls/v0.4.0-rc.99`.
- [ ] Create and publish release notes on Github.
- [ ] Publish the crates to crates.io