The Design and Formal Analysis of the Multi-Party Domain Cryptosystem (MPDC-I)

John G.Underhill

Quantum Resistant Cryptographic Solutions Corporation
contact@qrcscorp.ca

May 27, 2024

Abstract

The multi-party Domain Cryptosystem (MPDC-I) is a distributed key establishment and secure tunnel protocol that derives entropy from an MPDC Access Server together with a set of independent Agents. The protocol ensures that the session keys remain unpredictable as long as at least one Agent provides an uncompromised fragment. This paper gives a complete formal analysis of MPDC-I grounded in the behavior of the reference implementation and the engineering specification.

We introduce an execution model that captures the multi-party layout of the system, including certificate based authentication under the RDS root, topology distribution by the DLA, long term master fragment key establishment, and the fragment collection pipeline that precedes tunnel formation. The model defines adversarial capabilities through corruption oracles for Clients, MAS, Agents, and trust anchor services, and incorporates the valid time and sequence binding used in all authenticated network messaging.

Within this framework we prove a threshold style entropy lemma showing that a single honest Agent fragment contributes sufficient min entropy to the concatenated fragment string for the SHAKE based derivation to remain indistinguishable from random, even when all other fragments are adversarially chosen and all long term state except the uncompromised MFK is exposed.

Using these components, we give reduction based proofs for confidentiality, integrity, authentication, and post compromise security of the MPDC tunnel. The bounds depend on the collision and preimage resistance of SHAKE, the unforgeability of the signature scheme used in certificate validation and network control messaging, and the authenticity of the encrypted tunnel under RCS or AES-GCM with associated data formed from packet sequence and UTC timestamp. We also analyze attacks involving DLA compromise, certificate chain abuse, and topology desynchronization, and show that these affect availability but do not undermine tunnel secrecy or endpoint authentication.

These results demonstrate that MPDC-I provides strong security guarantees under standard assumptions, and that its distributed entropy design offers resilience against a broad range of corruption and compromise scenarios.

Contents

1 Introduction

The multi-party Domain Cryptosystem (MPDC-I) is a distributed key establishment and secure tunnel protocol designed for environments that require strong cryptographic guarantees in the presence of partial compromise. MPDC-I operates in a hierarchical trust domain anchored by an RDS root certificate authority and managed by a DLA service that distributes topology and certificate updates. Within this domain a Client, an MPDC Access Server (MAS), and a set of Agents collaborate to derive a high entropy session key that is later used to protect a bidirectional encrypted tunnel.

A defining feature of MPDC-I is its distributed entropy model. Each Agent generates a fragment of randomness and protects it using authenticated channels established through long term master fragment keys derived from an authenticated asymmetric key exchange. These keys are used to construct ephemeral fragment keys that mask Agent generated fragments, which are returned to the MAS and Client through the fragment collection pipeline. The Client and MAS reconstruct these fragments and combine them with a MAS generated value through a Keccak based derivation function. The resulting aggregate value determines the tunnel keys. The design ensures that the session key remains unpredictable as long as at least one Agent remains uncompromised, even if the remaining Agents behave adversarially. This provides resilience beyond standard two party authenticated key exchange, where compromise of either peer usually compromises the session.

The present work develops a complete formal security analysis of MPDC-I. The analysis includes a rigorous execution model that reflects the operational structure of the system, including certificate based authentication, topology synchronization, and the fragment collection process. To capture the multi-party nature of MPDC-I we define a dedicated MPDC Indistinguishability experiment with corruption oracles for Agents, Client, MAS, and trust anchor services. We introduce an entropy lemma that formalizes the claim that one honest Agent is sufficient to guarantee that the aggregate hash remains indistinguishable from random. Using this model we provide reduction based proofs of confidentiality, integrity, and post compromise security, and we derive concrete bounds in terms of the security of the signature scheme used for certificate authentication, the pseudo-randomness of SHAKE based derivation, and the security of the authenticated encryption layer. We also analyze the effects of DLA compromise and demonstrate that such compromise affects availability but does not enable an adversary to break authentication or recover tunnel keys.

1.1 Background and Motivation

Distributed key generation protocols and multi-party entropy constructions have been studied in several contexts, including threshold cryptography, contributory group key exchange, and robustness against partial compromise. MPDC-I differs from classical two party authenticated key exchange by relying on a set of independent Agents that contribute entropy through authenticated symmetric channels derived from long term master fragment keys. The system is intended for deployment in controlled domains where devices are provisioned under a common certificate hierarchy and where topology information is distributed through a central management service. These characteristics create a security setting that is not captured by standard two party AKE models and motivate the development of a protocol specific formal analysis.

1.2 Contributions

This paper provides the following contributions.

- An engineering level description of MPDC-I that reflects the normative specification and clarifies the operational behavior of Client, MAS, Agents, DLA, and RDS.
- A formal execution model that captures certificate validation, topology updates, authenticated master fragment key exchanges, fragment transmission, aggregation, and tunnel establishment.
- A dedicated MPDC Indistinguishability experiment that includes corruption oracles for Agents and trust anchors and models compromise scenarios specific to the protocol.
- An entropy lemma that proves that one honest Agent fragment provides sufficient min entropy for the aggregate hash to remain indistinguishable from random even against adversarial control of all other fragments and all long term state except the uncompromised MFK.
- Reduction based proofs of confidentiality, integrity, and post compromise security with concrete advantage bounds expressed in terms of the security of the signature scheme used for certificate authentication and the pseudo-randomness of SHAKE based derivation, together with the authenticity of the symmetric tunnel under RCS or AES-GCM.
- A trust anchor analysis showing that DLA compromise does not break authentication and that topology updates accepted by honest parties produce consistent and compatible views of the system.

1.3 Organization of the Paper

Section 2 gives an engineering level description of MPDC-I and summarizes the operational behavior of the protocol. Section 3 defines the execution model and the MPDC Indistinguishability experiment, including corruption oracles and admissibility conditions. Section 4 presents the entropy lemma for an honest Agent and the threshold view of the fragment system. Section 5 provides reduction based security proofs for confidentiality, integrity, and post compromise recovery. Section 6 analyzes trust anchors, topology consistency, and the effect of DLA compromise. Section 7 summarizes implementation assumptions and system considerations. Section 8 concludes with a summary of results and limitations.

2 Protocol Overview and Engineering Description

This section provides an engineering level description of MPDC I that follows the normative specification and prepares the ground for the formal model developed in later sections. The description introduces the entities in the system, their long term keys, their certificates, the topology and revocation model, the establishment of Master Fragment Keys, the fragment transmission pipeline, and the derivation and use of the tunnel keys.

2.1 Entities, Keys, and Trust Anchors

The MPDC domain contains a Client, an MPDC Access Server, and a set of Agents. Each device holds a long term EUF–CMA secure signature scheme signing key and a certificate issued under a single root certificate authority called the RDS. The RDS root key is embedded in all devices and serves as the trust anchor for the entire domain. A secondary management authority called the DLA holds a certificate signed by the RDS and distributes topology information, certificate lists, and revocation updates. Devices accept topology or certificate changes only when the associated message verifies under a DLA certificate that chains to the RDS root.

Each device certificate contains a public verification key, an issuer, a serial number, a validity interval, and a configuration identifier that binds the certificate to a particular cryptographic configuration set. The configuration identifier is included in all certificate hashes and in key derivation functions to prevent downgrade attacks caused by inconsistent parameter sets.

2.2 Topology, Certificates, and Revocation

The DLA is responsible for broadcasting topology updates that describe the set of active Agents. Each topology message contains a sequence number and a timestamp and is signed under the DLA signing key. Devices validate these messages under the RDS root key and adopt the topology only if the message is newer than their current view and the timestamps lie within the permitted clock window. This ensures convergence of topology across the domain.

When a certificate is revoked the DLA broadcasts a signed revocation mes sage that contains the serial number of the revoked certificate and the configuration set in which it appears. Devices remove the corresponding certificate from their local trust store. If a device loses its own certificate due to expiry or revocation it cannot participate in MFK exchange until a new certificate is provisioned. No device may accept a certificate with an expiration time that exceeds that of the RDS root.

2.3 Master Fragment Key Establishment

Master Fragment Keys are shared values that pairwise link the Client, the MAS, and each Agent. These values are established through authenticated in vocations of the IND-CCA secure KEM. Before a KEM exchange each party verifies the peer certificate under the RDS root and checks that the configuration identifier matches its own.

For each Agent there are two MFK values: one shared between the Client and the Agent and one shared between the MAS and the Agent. The Client or MAS generates a KEM keypair, sends the public key to the Agent in a signed message, and receives a KEM ciphertext from the Agent that encapsulates a shared secret. The corresponding MFK is defined as the KEM shared secret. All authenticated messages in this exchange include a timestamp and a sequence number to prevent replay.

2.4 Fragment Generation and Delivery

Each Agent generates a 256 bit fragment of randomness using a cryptographically secure random number generator. The Agent derives two Ephemeral Fragment Keys from its MFK values using SHAKE based key derivation. One EFK is used to protect the fragment on the path to the MAS and the other is used to protect the fragment on the path to the Client.

The Agent masks the fragment with the EFK by applying an XOR against a SHAKE derived keystream. The Agent also computes a KMAC tag over the masked fragment and a signed header that includes the Agent certificate hash, the MAS or Client certificate hash, and the MAS token. The Agent sends the protected fragment to the MAS and to the Client in two independent authenticated messages. Agents do not learn any tunnel keys and do not exchange fragments with each other.

2.5 Aggregation and Tunnel Key Derivation

The MAS and the Client reconstruct fragments by verifying the signature signed headers, verifying the KMAC tags under their respective EFK keys, and removing the masking. The MAS adds its own fragment to the set of reconstructed fragments. Once all expected fragments are available both parties absorb them into a Keccak based state in a fixed order determined by the topology. The output of the final SHAKE operation is the aggregate value h which serves as the input to the tunnel key derivation function.
Transmit and receive keys are derived from h using separate SHAKE invocations. The Client derives keys $(K_{\text{tx}}, K_{\text{rx}})$ and the MAS derives $(K_{\text{rx}}, K_{\text{tx}})$ so that the two parties share forward and backward channel keys. These keys are used in RCS authenticated encryption for all bidirectional tunnel traffic.

2.6 Encrypted Tunnel and Replay Protection

Tunnel messages are protected by RCS used in an Encrypt then MAC construction. Each tunnel message contains a header with a function code, a ciphertext length, a sequence number, and a timestamp. The header is included in the associated data of the AEAD so that any modification causes the tunnel to terminate.
Sequence numbers increase by one for each message in each direction. Times tamps are validated against a local clock with a fixed time window. A message is accepted only if the timestamp lies within the permitted window, the sequence number matches the expected value, and the authentication tag verifies. Reception of an invalid message or a deviation in sequence or timestamp results in immediate teardown of the tunnel.

2.7 Lifecycle, Renewal, and Failure Modes

Certificates expire at regular intervals defined by the root certificate. When a certificate nears expiry the DLA distributes renewal information and devices provision new certificates through an authenticated management process. When the topology changes due to revocation or addition of Agents the DLA broad casts a new signed topology message. Devices update their view of the active

Agent set only after verifying the DLA signature and confirming that the up date is monotonic.

If an Agent is unavailable during fragment collection the Client and MAS treat it as absent for that round. A tunnel key is generated whenever at least one Agent contributes a valid fragment. If no Agents are available the system can not generate a session key and the Client and MAS do not initiate a tunnel. In the event of DLA compromise the protocol continues to provide confidentiality and authentication for tunnel traffic although topology and update distribution may become unreliable.

3 Network Message Exchanges

This section describes the network message exchanges that implement the MPDC control plane and tunnel management. Messages are grouped by the primary service role that participates in or coordinates the exchange. For each group, the subsections define the logical request–response structure, the devices that send and receive the messages, the cryptographic operations applied to each packet, and the role of the exchange within the overall MPDC framework. Formal message equations and detailed cryptographic bindings will be introduced in later subsections.

Message	**Direction**	**Purpose**
mfk request	Initiator to Agent	Begin MFK exchange
mfk response	Agent to Initiator	Return authenticated pk
mfk establish	Initiator to Agent	Send KEM encapsulation
fragment query	MAS to Agent	Request encrypted fragments
fragment reply	Agent to MAS	Return MAS and Client fragments
fragment query	Client to Agent	Client directed fragment request
fragment reply	Agent to Client	Return encrypted Client fragment
converge	DLA to All	Topology broadcast
converge ack	Node to DLA	Acknowledge topology update
revoke	DLA to All	Certificate or Agent revocation
join	Node to DLA	Request to join domain
resign	Node to DLA	Graceful removal from domain
error	Any to Any	Protocol error notification

Table 1: MPDC network message types and their roles within the protocol.

3.1 RDS Message Exchanges

The Root Domain Security (RDS) service acts as the root trust anchor for all devices. It issues and renews certificates and authorizes revocation events that are propagated through the DLA and MAS. From the perspective of the network layer, RDS participates in certificate signing exchanges that are proxied by the DLA.

3.1.1 Remote Certificate Signing Exchange

The remote certificate signing exchange allows the DLA to obtain an RDS root signature on a child certificate on behalf of a remote device. In this exchange the DLA acts as the requester and the RDS as the responder. The DLA constructs a packet that binds a local network time stamp and sequence number to the serialized child certificate, authenticates this binding under its own signing key, and sends the resulting message to the RDS. The RDS verifies the DLA signature under the DLA certificate anchored in the RDS trust store. If verification and policy checks succeed, the RDS signs the certificate under the root key and returns the root signed certificate to the DLA, which verifies the root signature and installs the updated certificate in its local store. No step requires the DLA to sign a certificate that already carries an RDS signature; the DLA signs only the request transcript, and the RDS signs only the certificate.

Formally, let D denote the DLA, R the RDS, ts a network time stamp and sequence pair encoded in the MPDC subheader, cert the unsigned child certificate to be promoted, and scert the same certificate after root signing. Let $H(\cdot)$ be the message hash over the concatenation of the subheader and the serialized certificate, and let $\mathrm{Sign}_{\mathsf{sk}_D}$ and $\mathrm{Sign}_{\mathsf{sk}_R}$ denote signatures under the DLA and RDS signing keys respectively. The request and response have the form: Let $\mathsf{subhdr}(ts) \in \{0,1\}^{\texttt{MPDC\_PACKET\_SUBHEADER\_SIZE}}$ denote the fixed width serialization of the packet time and sequence, and let cert be the serialized child certificate. The DLA request has the form

$$D \to R: \quad m_{\mathsf{req}} = \big(\mathsf{subhdr}(ts),\ \mathsf{cert},\ \sigma_D\big),\ \sigma_D = \mathrm{Sign}_{\mathsf{sk}_D}\big(H(\mathsf{subhdr}(ts) \parallel \mathsf{cert})\big).$$

where the implementation copies the subheader and the serialized certificate into the message buffer then appends σ_D as a fixed width signed hash. Upon receipt, the RDS verifies σ_D against the stored DLA certificate, applies the root signing operation:

$$\mathsf{scert} = \mathrm{RootSign}_R(\mathsf{cert}),$$

and constructs the response:

$$R \to D: \quad m_{\mathsf{resp}} = \big(ts',\ \mathsf{scert}\big),$$

where ts' is a fresh subheader time and sequence value for the response and scert is serialized into the message body. On receipt, the DLA verifies σ_R with the RDS certificate anchored under the RDS root and checks that the subheader fields reconstructed from the packet header match those covered by the signature. If both checks succeed, the DLA replaces its local copy of cert with scert, completing the remote signing operation in the MPDC trust framework.

3.1.2 Root Certificate Status and Epoch Transitions

The RDS root certificate defines the global trust epoch for all MPDC devices. Although the network layer does not exchange explicit epoch transition messages, the RDS epoch is implicitly enforced through certificate validity checks, signature verification, and the acceptance or rejection logic implemented in the DLA, MAS, and Client network functions. Each certificate issued under the RDS carries an epoch identifier and validity interval. When the RDS root key is rotated, the epoch value increments and all dependent certificates must be refreshed or replaced. Devices enforce these constraints during every authenticated message exchange.

Let ep_R denote the current root epoch, stored in the trusted local RDS certificate. Each subordinate certificate cert_i contains an epoch field ep_i and validity window $[t_i^{\min}, t_i^{\max}]$. A device accepts a received certificate or signature only if the following checks succeed:

$$\mathsf{ep}_i = \mathsf{ep}_R, \qquad t_i^{\min} \le t_{\mathsf{now}} \le t_i^{\max}, \qquad \mathrm{Verify}_{\mathsf{pk}_R}(\sigma_i, H(\mathsf{cert}_i)) = \mathsf{true}.$$

The network code enforces this implicitly in all verification functions by loading the root certificate, checking the epoch equality, and validating the signature using the RDS public key.

Epoch transitions occur only when the RDS root key is rotated. Conceptually, the transition from epoch ep_R to $\mathsf{ep}_R + 1$ is a deterministic state update:

$$\mathsf{ep}_R^{\mathrm{new}} = \mathsf{ep}_R^{\mathrm{old}} + 1, \qquad \mathsf{pk}_R^{\mathrm{new}}, \mathsf{sk}_R^{\mathrm{new}} = \mathrm{KeyGen}(),$$

followed by re-signing or re-issuing all subordinate certificates under the new root key. No network-visible packet triggers this transition; rather, the DLA and MAS detect the change when the stored RDS certificate is updated or replaced during system provisioning.

Let $\mathsf{CertStore}(d)$ denote the certificate store of device d. A valid epoch transition requires the following local update:

$$\mathsf{CertStore}(d) \leftarrow \left\{\mathrm{RootCert}(\mathsf{pk}_R^{\mathrm{new}}, \mathsf{ep}_R^{\mathrm{new}}),\ \mathrm{ReissuedChildren}(\mathsf{pk}_R^{\mathrm{new}}, \{\mathsf{cert}_i\})\right\}.$$

All inbound messages whose certificate epochs do not match $\mathsf{ep}_R^{\text{new}}$ are rejected. This prevents replay of stale messages, prevents signature acceptance under prior epochs, and ensures that every authenticated exchange is anchored to the current RDS trust domain.
Although implicit, these epoch transitions define the acceptance boundary for every MPDC message type. All request–response message sets described in later sections derive their validity from the epoch equality predicate and the root signature carried in the corresponding certificates.

3.2 DLA Message Exchanges

3.2.1 Topology Announcement Broadcast

The topology announcement broadcast allows the DLA to distribute an authenticated snapshot of the network topology to all Clients, MAS instances, and Agents. This message provides each device with the current topology epoch, the list of active nodes, and the DLA's authoritative view of adjacency and certificate status. Because this message is broadcast rather than point-to-point, its authenticity and integrity rely entirely on the DLA signature bound to the announcement transcript.

Let D denote the DLA, and let $\mathcal{R} = \{C, M, A_1, \dots, A_n\}$ denote the set of all reachable devices (Clients, MAS instances, and Agents). Let ep_T be the current topology epoch maintained by the DLA, and let Topo be the serialized topology vector encoding active nodes, adjacency records, certificate serials, and status flags as constructed in the code. The DLA constructs a broadcast packet of the form:

$$m_{\mathsf{ann}} = \big(\mathsf{ep}_T,\ \mathsf{Topo},\ \sigma_D\big),$$

where:

$$\sigma_D = \mathrm{Sign}_{\mathsf{sk}_D}\big(H(\mathsf{ep}_T \parallel \mathsf{Topo})\big).$$

The hash $H(\cdot)$ mirrors the internal logic in the network layer: the DLA writes the MPDC header, appends the topology epoch and topology bytes, then computes a fixed width hash over this region prior to signing. The network code verifies that the signature covers both the topology epoch and the exact serialized byte sequence of the topology vector.
The broadcast semantics are therefore

$$D \longrightarrow \mathcal{R} : \quad m_{\mathsf{ann}}.$$

Upon receiving m_{ann}, each device $r \in \mathcal{R}$ performs:

1. **Signature verification**

$$\mathsf{Verify}_{\mathsf{pk}_D}\big(\sigma_D,\ H(\mathsf{ep}_T \parallel \mathsf{Topo})\big) = \mathsf{true},$$

using the DLA certificate anchored under the RDS root.

2. **Epoch comparison**

$$\mathsf{ep}_T > \mathsf{ep}_T^{(r)} \quad \Rightarrow \quad \text{accept and update.}$$

3. **Topology store update**

$$\mathsf{TopoStore}(r) \leftarrow \mathsf{Topo}, \quad \mathsf{ep}_T^{(r)} \leftarrow \mathsf{ep}_T.$$

If any of these checks fail, the device discards the announcement.
The topology announcement broadcast therefore provides a one-to-many authenticated dissemination mechanism:

$$D \to \{C, M, A_i\}_i,$$

ensuring that all nodes converge to a consistent, DLA-authoritative topology snapshot. This broadcast is foundational to the MPDC trust and routing framework: all subsequent request–response exchanges rely on the topology epoch and certificate state established through this mechanism.

3.2.2 Network Convergence Exchange

The network convergence exchange is an administrative procedure initiated by the DLA to synchronize its topology and certificate database with the actual state of each MAS and Agent. From the perspective of the message layer, the DLA acts as the requester and each MAS or Agent acts as a responder. For a given remote node identifier, the DLA sends the serialized copy of its stored topology node entry to the remote device, authenticated under the DLA signing key. The remote device compares this entry to its local topology state and either confirms that the entries match or, in the intended design, returns an updated certificate and node description that the DLA can install. The current implementation realizes the confirmation path and reserves explicit update packets for future use.

Let D denote the DLA, let X denote a MAS or Agent, and let id be the identifier of X in the DLA topology. Write:

$$\mathsf{node}_D(\mathsf{id}) \in \{0,1\}^{\ell_{\mathsf{node}}} \quad \text{and} \quad \mathsf{node}_X(\mathsf{id}) \in \{0,1\}^{\ell_{\mathsf{node}}}$$

for the serialized node entries stored at the DLA and at device X, respectively, using the fixed width encoding implemented by `mpdc_topology_node_serialize`. Let ts denote the network time stamp and sequence subheader, and let $H(\cdot)$ be the message hash used by `mpdc_certificate_message_hash_sign`. Signatures under the DLA and device signing keys are written $\mathrm{Sign}_{\mathsf{sk}_D}$ and $\mathrm{Sign}_{\mathsf{sk}_X}$.

Convergence request. For each remote node id, the DLA constructs a convergence request packet by writing the subheader ts into the message buffer, then appending the serialized node $\mathsf{node}_D(\mathsf{id})$, and finally appending a signed hash of this transcript:

$$D \to X : \quad m_{\mathsf{req}}(\mathsf{id}) = \big(\mathsf{ts},\ \mathsf{node}_D(\mathsf{id}),\ \sigma_D\big),$$

where:

$$\sigma_D = \mathrm{Sign}_{\mathsf{sk}_D}\big(H\big(\mathsf{ts} \parallel \mathsf{node}_D(\mathsf{id})\big)\big).$$

This is implemented by `network_converge_request_packet`, which calls `network_subheader_serialize` to encode ts, copies $\mathsf{node}_D(\mathsf{id})$ into the packet, and invokes `network_message_hash_sign` with the DLA signing key `state->sigkey`. The on wire header uses the convergence request flag and sequence value `NETWORK_CONVERGE_REQUEST_SEQUENCE`.
On receipt, the responder X uses `network_converge_request_verify` to validate the request. This function first checks the header fields, then calls `network_message_signed_hash_verify` with the DLA certificate `state->rcert` to recover the serialized node $\mathsf{node}_D(\mathsf{id})$ and verify σ_D. It then serializes its own local node $\mathsf{node}_X(\mathsf{id})$ and compares the two byte strings.

Convergence response (confirmation path). If the local topology entry matches the DLA view, that is

$$\mathsf{node}_X(\mathsf{id}) = \mathsf{node}_D(\mathsf{id}),$$

then the device constructs a confirmation response that mirrors the request structure, but signed under the device key:

$$X \to D : \quad m_{\mathsf{resp}}(\mathsf{id}) = \big(\mathsf{ts}',\ \mathsf{node}_X(\mathsf{id}),\ \sigma_X\big),$$

where

$$\sigma_X = \mathrm{Sign}_{\mathsf{sk}_X}\big(H\big(\mathsf{ts}' \parallel \mathsf{node}_X(\mathsf{id})\big)\big).$$

This is realized by `network_converge_response_packet`, which uses `network_message_hash_sign` with the responder signing key `state->sigkey` and message $\mathsf{node}_X(\mathsf{id})$. The public interface `mpdc_network_converge_response` calls `network_converge_request_verify` to check the request, then builds and transmits this response over the socket `state->csock` with the convergence response header flag and sequence `NETWORK_CONVERGE_RESPONSE_SEQUENCE`. At the DLA, `network_converge_response_verify` validates the response header, then calls `network_message_signed_hash_verify` with the responder certificate `state->rcert` to recover $\mathsf{node}_X(\mathsf{id})$ and verify σ_X. It serializes the DLA copy $\mathsf{node}_D(\mathsf{id})$ and checks equality:

$$\mathsf{node}_X(\mathsf{id}) = \mathsf{node}_D(\mathsf{id}) \quad \Rightarrow \quad \text{synchronized}(\mathsf{id}) := \mathsf{true},$$

otherwise it reports `mpdc_protocol_error_node_not_found`. This confirmation path is the behavior exercised in `network_test_converge`.

Convergence update (intended path). The convergence comments in `network.c` and the reserved size constant `NETWORK_CONVERGE_UPDATE_MESSAGE_SIZE` describe an additional certificate update case. Conceptually, when

$$\mathsf{node}_X(\mathsf{id}) \neq \mathsf{node}_D(\mathsf{id}),$$

the device should serialize its current certificate cert_X and construct an update message of the form

$$X \to D: \quad m_{\mathsf{upd}}(\mathsf{id}) = \big(\mathsf{ts}'', \mathsf{cert}_X, \sigma'_X\big), \qquad \sigma'_X = \mathrm{Sign}_{\mathsf{sk}_X}\big(H(\mathsf{ts}'' \parallel \mathsf{cert}_X)\big),$$

summarized in the code comments as

$$\mathsf{rcert} = \big(V_{\mathsf{root}}(\mathsf{rcert}),\ V_{\mathsf{rcert}}(H(\mathsf{ts} \mid \mathsf{rcert}))\big),$$

$$\mathsf{rnode} = \begin{cases} M(\mathsf{node}, \mathrm{Sign}(H(\mathsf{ts} \mid \mathsf{node}))) \to D, & \text{if entries match,} \\ M(\mathsf{cert}, \mathrm{Sign}(H(\mathsf{ts} \mid \mathsf{cert}))) \to D, & \text{if entries differ.} \end{cases}$$

The DLA would then verify σ'_X using cert_X under the RDS root, purge the stale node and certificate from its database, and install the updated pair.

In the current implementation, the public convergence interface uses the confirmation path and the equality check on serialized nodes to mark entries as synchronized. The presence of `NETWORK_CONVERGE_UPDATE_MESSAGE_SIZE` and the convergence comments define the intended extension to a three message, state indexed exchange in which the DLA first queries the topology state at X, then either records a confirmation or applies a certified update to its own copy of the topology and certificate data.

3.2.3 Incremental Certificate Update Exchange

Incremental certificate updates allow the DLA or a delegated management node to fetch the current public certificate from a remote device using only its certificate serial. In this exchange the requester D (typically the DLA) sends a compact query containing a certificate serial to a remote host X. The remote host checks that the serial matches its own local certificate, then returns a signed update packet that encapsulates the current certificate together with a time stamp and sequence number. The response is authenticated under the device key and anchored to the RDS root, and the time and sequence values are bound both into the packet header and into the signed message body, which prevents replay of stale certificate updates.

Let $\mathsf{serial} \in \{0,1\}^{\ell_{\mathsf{serial}}}$ denote a certificate serial, where:
$\ell_{\mathsf{serial}} = \texttt{MPDC\_CERTIFICATE\_SERIAL\_SIZE}$.

Let cert_X be the child certificate of device X with serial serial, and let $\mathsf{ts} = (\mathsf{seq}, \mathsf{time})$ be the pair of sequence number and UTC time stored in the network packet header. The incremental update request is generated by `network_incremental_update_request_packet`, which creates a packet header with fixed sequence value `NETWORK_INCREMENTAL_UPDATE_REQUEST_SEQUENCE`, sets the creation time, and copies the requested serial into the message body:

$$D \to X: \quad m_{\mathsf{req}}(\mathsf{serial}) = \big(\mathsf{flag} = \texttt{incremental\_update\_request},\ \mathsf{seq} = s_{\mathsf{req}}, \\ \mathsf{time} = t_{\mathsf{req}},\ \mathsf{serial}\big).$$

with:

$$\begin{aligned} |\mathsf{serial}| &= \ell_{\mathsf{serial}}, \\ |\mathsf{body}| &= \texttt{NETWORK\_INCREMENTAL\_UPDATE\_REQUEST\_MESSAGE\_SIZE} \\ &= \texttt{MPDC\_CERTIFICATE\_SERIAL\_SIZE}. \end{aligned}$$

The public function `mpdc_network_incremental_update_request` connects to the remote address recorded in the topology node, sends this packet of total size `NETWORK_INCREMENTAL_UPDATE_REQUEST_PACKET_SIZE`, and then waits for an update response.

On the remote side, `mpdc_network_incremental_update_response` receives the request into $\mathsf{packet}_{\mathsf{in}}$, and `network_incremental_update_response_packet` first checks that the requested serial matches the local certificate serial:

$$\mathsf{packet}_{\mathsf{in}}.\mathsf{body}[0..\ell_{\mathsf{serial}} - 1] = \mathsf{serial}(\mathsf{cert}_X).$$

If this equality holds, it constructs a response packet with flag `incremental_update_response`, sequence `NETWORK_INCREMENTAL_UPDATE_RESPONSE_SEQUENCE`.

The response body is created by `network_certificate_hash_sign`, which writes the packet subheader and certificate into the message, then signs the resulting transcript. Let

$$\mathsf{ts}' = (\mathsf{seq}', \mathsf{time}') = \big(\mathsf{packet}_{\mathsf{out}}.\mathsf{sequence}, \mathsf{packet}_{\mathsf{out}}.\mathsf{utctime}\big)$$

be the sequence and time from the response header. The subheader serialization function

$$\begin{aligned}\mathsf{subhdr}(\mathsf{ts}') &= \texttt{network\_subheader\_serialize}(\mathsf{ts}') \in \{0,1\}^{\ell_{\mathsf{sub}}},\\ \ell_{\mathsf{sub}} &= \texttt{MPDC\_PACKET\_SUBHEADER\_SIZE}.\end{aligned}$$

writes seq' and time' into a fixed width byte array. The responder then constructs

$$M_X = \mathsf{subhdr}(\mathsf{ts}') \parallel \mathsf{cert}_X \in \{0,1\}^{\ell_{\mathsf{sub}}+\ell_{\mathsf{cert}}}$$

and computes a signed hash

$$\sigma_X = \mathrm{Sign}_{\mathsf{sk}_X}\big(H(M_X)\big),$$

where $\mathrm{Sign}_{\mathsf{sk}_X}$ and H are instantiated by `mpdc_certificate_message_hash_sign`. The complete incremental update response is

$$X \to D: \quad m_{\mathsf{resp}}(\mathsf{serial}) = \big(\mathsf{flag},\ \mathsf{seq}',\ \mathsf{time}',\ \mathsf{subhdr}(\mathsf{ts}'),\ \mathsf{cert}_X,\ \sigma_X\big).$$

with $\mathsf{subhdr}(\mathsf{ts}')$ and cert_X occupying the first

$$\texttt{NETWORK\_CERTIFICATE\_UPDATE\_SIZE} = \ell_{\mathsf{sub}} + \ell_{\mathsf{cert}}$$

bytes of the message body.
At the requester, `network_incremental_update_verify` first validates the header using `network_header_validate` with the expected flag, sequence, message length and the time window check `mpdc_packet_time_valid`. It then calls `network_certificate_signed_hash_verify`, which performs the following steps:

1. Verify the signature over the signed part of the body:

   $$\mathrm{Verify}_{\mathsf{pk}_X}\big(\sigma_X,\ H(M_X)\big) = \mathsf{true},$$

   where pk_X is extracted from cert_X and validated under the RDS root.

2. Recompute the subheader from the received header:

$$\mathsf{subhdr}'(\mathsf{ts}') = \texttt{network\_subheader\_serialize}(\mathsf{packet}_{\mathsf{in}}),$$

   and check that

$$\mathsf{subhdr}'(\mathsf{ts}') = \mathsf{subhdr}(\mathsf{ts}'),$$

   which enforces equality between the header time and sequence values and the values that were included in the signed transcript.

3. Deserialize cert_X from the body and run `mpdc_network_certificate_verify` against the configured root.

If all checks succeed, the requester treats the mapping

$$\mathsf{serial} \longmapsto \mathsf{cert}_X$$

as an authenticated incremental update and installs cert_X in its local certificate cache. Any attempt to replay an old response with modified header time or sequence fails the subheader comparison, and any attempt to replay an expired packet with intact header fields fails the `mpdc_packet_time_valid` check. The incremental update exchange therefore realizes a minimal request–response mapping from certificate serial identifiers to fresh certificate payloads, with time and sequence values cryptographically bound into both the packet header and the signed body.

3.2.4 Agent Join Registration Exchange

The Agent join registration exchange allows a new Agent A to introduce itself to the MPDC domain and register its certificate and topology metadata with the DLA D. The Agent is the requester and the DLA is the responder. The Agent constructs a join request packet containing its child certificate, a serialized topology node entry describing its network position, and a signed hash that binds these fields to the packet time stamp and sequence number. The DLA verifies the certificate chain under the RDS root, validates that topology constraints are satisfied, and, on success, returns a join response containing a signed confirmation record. This process ensures that only authenticated and topology-consistent Agents can join the MPDC domain.

Let cert_A denote the Agent child certificate, node_A the serialized topology node produced by `mpdc_topology_node_serialize`, and $\mathsf{ts} = (\mathsf{seq}, \mathsf{time})$ the packet sequence number and network time. Let $H(\cdot)$ be the MPDC message hash and $\mathrm{Sign}_{\mathsf{sk}_A}$ and $\mathrm{Sign}_{\mathsf{sk}_D}$ be signatures under the Agent and DLA signing keys. The corresponding network flags are `join_request` and `join_response`, implemented with `NETWORK_JOIN_REQUEST_SEQUENCE` and `NETWORK_JOIN_RESPONSE_SEQUENCE`.

Join request. The Agent constructs the join request with body

$$M_A = \mathsf{subhdr}(\mathsf{ts}) \parallel \mathsf{cert}_A \parallel \mathsf{node}_A,$$

where $\mathsf{subhdr}(\mathsf{ts})$ is the fixed width serialization of ts defined by `network_subheader_serialize`. The Agent signs the hash of this transcript:

$$\sigma_A = \mathrm{Sign}_{\mathsf{sk}_A}\big(H(M_A)\big).$$

The complete join request message is therefore

$$A \to D : \quad m_{\mathsf{join\_req}} = \big(\mathsf{flag},\, \mathsf{seq},\, \mathsf{time},\, \mathsf{cert}_A,\, \mathsf{node}_A,\, \sigma_A\big),$$

On receipt, the DLA executes `network_join_request_verify`, which:

1. Validates the header fields, including time and sequence.
2. Recovers cert_A and node_A from the body.
3. Verifies
   $$\mathrm{Verify}_{\mathsf{pk}_A}(\sigma_A,\, H(M_A)) = \mathsf{true},$$
   where pk_A is extracted from cert_A.
4. Applies `mpdc_network_certificate_verify` to validate the signature chain of cert_A under the RDS root.
5. Ensures the topology node is consistent with the DLA topology constraints.

Join response. If the checks succeed, the DLA constructs a join response packet containing the Agent certificate, the DLA's corresponding topology view $\mathsf{node}_D(A)$, and a signed transcript binding these to a fresh subheader time stamp and sequence number ts':

$$M_D = \mathsf{subhdr}(\mathsf{ts}') \parallel \mathsf{cert}_A \parallel \mathsf{node}_D(A).$$

The DLA signature is:

$$\sigma_D = \mathrm{Sign}_{\mathsf{sk}_D}\big(H(M_D)\big).$$

The join response is:

$$D \to A : \quad m_{\mathsf{join\_resp}} = \big(\mathsf{flag},\, \mathsf{seq}',\, \mathsf{time}',\, \mathsf{cert}_A,\, \mathsf{node}_D(A),\, \sigma_D\big),$$

On receipt, the Agent verifies: 1. Header time and sequence. 2. DLA signature under the DLA certificate (anchored to RDS). 3. Consistency of the DLA topology entry with the expected placement of the Agent.

Exchange summary. The Agent join registration exchange therefore realizes a two message authenticated transcript:

$$A \xrightarrow{\texttt{join\_request}} D \quad \text{and} \quad D \xrightarrow{\texttt{join\_response}} A,$$

with both messages binding the sender identity, the certificate payload, the topology node, and the packet time and sequence number through a signed hash. The certificate chain is verified under the RDS root at both endpoints, and the exchange establishes the Agent as an authenticated topology participant in the MPDC domain.

3.2.5 Server and Client Join Update Exchange

The join update exchange allows MAS servers and Clients to register or refresh their status with the DLA. Unlike the Agent join exchange, which introduces a new topology node, the join update exchange operates on an existing DLA topology entry indexed by the sender certificate serial. The purpose of the exchange is to confirm continued presence, update addressing fields, and refresh certificate–topology bindings while ensuring that the DLA's view of the network remains synchronized with all participating endpoints.
Let $X \in \{\text{Client}, \text{MAS}\}$ denote the requesting device, and let D denote the DLA. Let cert_X be the child certificate of X, node_X a serialized topology node expressing the updated addressing or role metadata, and $\mathsf{ts} = (\mathsf{seq}, \mathsf{time})$ the header time and sequence number. As in the Agent join exchange, a signed hash binds these fields to the sender.
The network flags for this exchange are
`join_update_request` and `join_update_response`.

Join update request. Device X constructs the join update request body

$$M_X = \mathsf{subhdr}(\mathsf{ts}) \parallel \mathsf{cert}_X \parallel \mathsf{node}_X$$

and signs it using its device signing key:

$$\sigma_X = \mathrm{Sign}_{\mathsf{sk}_X}\big(H(M_X)\big).$$

The complete request is:

$$X \to D : \quad m_{\mathsf{upd\_req}} = \big(\mathsf{flag},\ \mathsf{seq},\ \mathsf{time},\ \mathsf{cert}_X,\ \mathsf{node}_X,\ \sigma_X\big),$$

Upon receiving the request, the DLA performs:

1. **Header validation** (flag, size, sequence, time).
2. **Signature verification**

$$\mathrm{Verify}_{\mathsf{pk}_X}(\sigma_X,\ H(M_X)) = \mathsf{true}.$$

3. **Certificate verification** under the RDS root.
4. **Topology validation**: The DLA locates the topology entry indexed by $\mathsf{serial}(\mathsf{cert}_X)$. If none exists, the DLA may treat the request as a late join and insert a new entry. If an entry exists, its fields are updated according to the values provided in node_X.

Topology state update. Let $\mathsf{id} = \mathsf{serial}(\mathsf{cert}_X)$. Let $\mathsf{node}_D(\mathsf{id})$ be the DLA's stored topology node.
A successful join update request induces the update rule:

$$\mathsf{node}_D(\mathsf{id}) \leftarrow \mathrm{UpdateFields}\big(\mathsf{node}_D(\mathsf{id}), \mathsf{node}_X\big),$$

where UpdateFields overwrites only those fields permitted by the MPDC topology specification (for example, address, port, and role indicators). This preserves the invariant that certificate and topology indices remain consistent across the network.

The DLA marks the node as refreshed:

$$\mathrm{LastSeen}(\mathsf{id}) \leftarrow t_{\mathsf{req}}.$$

Join update response. After updating its topology state, the DLA prepares a join update response. Let $\mathsf{node}'_D(\mathsf{id})$ be the updated node. The DLA assembles:

$$M_D = \mathsf{subhdr}(\mathsf{ts}') \parallel \mathsf{cert}_X \parallel \mathsf{node}'_D(\mathsf{id}),$$

where $\mathsf{ts}' = (\mathsf{seq}', \mathsf{time}')$ is fresh. The response signature is:

$$\sigma_D = \mathrm{Sign}_{\mathsf{sk}_D}\big(H(M_D)\big).$$

The message is then:

$$D \to X : \quad m_{\mathsf{upd\_resp}} = \big(\mathsf{flag},\ \mathsf{seq}',\ \mathsf{time}',\ \mathsf{cert}_X,\ \mathsf{node}'_D(\mathsf{id}),\ \sigma_D\big),$$

On receipt, device X validates the DLA signature and replaces its local DLA topology view for id with the supplied $\mathsf{node}'_D(\mathsf{id})$.

Exchange summary. The join update exchange is thus an authenticated two message protocol:

$$X \xrightarrow{\texttt{join\_update\_request}} D \quad \text{and} \quad D \xrightarrow{\texttt{join\_update\_response}} X.$$

Its key properties are:

1. **State dependence**: The DLA response depends explicitly on the prior topology entry associated with $\mathsf{serial}(\mathsf{cert}_X)$.
2. **Topology mutation**: The DLA updates only permissible node metadata fields while maintaining certificate–node index consistency.
3. **Authenticated transcript**: Both messages bind time, sequence, certificate, and topology content through a signed hash, preventing replay or spoofing.
4. **Integration with convergence**: Join updates serve as lightweight local corrections to the DLA topology store, while the convergence exchange provides full synchronization across all nodes.

This exchange therefore ensures that Clients and MAS servers remain correctly registered in the MPDC domain and that the DLA's topology view reflects the current operational state of all participating entities.

3.2.6 Keep Alive Exchange

The keep alive exchange enables Clients, MAS instances, and Agents to maintain a liveness relationship with the DLA. Each device X periodically sends a keep alive request that binds its certificate identity and a fresh time–sequence pair into an authenticated message. The DLA validates the request, updates the device's liveness status, and returns a keep alive response containing a new authenticated time–sequence pair. Both directions prevent spoofed liveness by authenticating the sender through a signed hash over a transcript that includes the packet metadata.
The exchange uses the flags `keepalive_request` and `keepalive_response`. Let $H(\cdot)$ be the message hash, $\mathsf{ts} = (\mathsf{seq}, \mathsf{time})$ the time–sequence subheader, and $\mathrm{Sign}_{\mathsf{sk}_X}$, $\mathrm{Sign}_{\mathsf{sk}_D}$ denote signatures under device X and DLA signing keys, respectively.

Keep alive request. Device X constructs the request transcript

$$M_X = \mathsf{subhdr}(\mathsf{ts}) \parallel \mathsf{cert}_X,$$

where $\mathsf{subhdr}(\mathsf{ts})$ is the fixed width encoding of the packet time and sequence produced by `network_subheader_serialize`, and cert_X is the sender certificate (or, in compact modes, its serial and role fields).
The device signs the transcript hash:

$$\sigma_X = \mathrm{Sign}_{\mathsf{sk}_X}\big(H(M_X)\big).$$

The complete request is

$$X \to D: \quad m_{\mathsf{ka\_req}} = \big(\mathsf{flag},\ \mathsf{seq} = s_{\mathsf{req}},\ \mathsf{time} = t_{\mathsf{req}},\ \mathsf{cert}_X,\ \sigma_X\big),$$

Upon receiving the request, the DLA:

1. Validates header time and sequence using `mpdc_packet_time_valid`.
2. Verifies
$$\mathrm{Verify}_{\mathsf{pk}_X}(\sigma_X,\ H(M_X)) = \mathsf{true}.$$
3. Validates cert_X under the RDS root.
4. Updates the liveness timestamp for $\mathsf{serial}(\mathsf{cert}_X)$:
$$\mathrm{LastSeen}(\mathsf{serial}_X) \leftarrow t_{\mathsf{req}}.$$

Keep alive response. The DLA constructs its own transcript using a fresh subheader:

$$M_D = \mathsf{subhdr}(\mathsf{ts}') \parallel \mathsf{cert}_X,$$

and signs the transcript using its signing key:

$$\sigma_D = \mathrm{Sign}_{\mathsf{sk}_D}\big(H(M_D)\big).$$

The response is:

$$D \to X: \quad m_{\mathsf{ka\_resp}} = \big(\mathsf{flag},\ \mathsf{seq}',\ \mathsf{time}',\ \mathsf{cert}_X,\ \sigma_D\big),$$

Device X verifies the DLA signature under pk_D, checks the time–sequence freshness of the response, and updates its local DLA liveness state.

Monotone sequence and bounded skew. Both sides must satisfy:
1. **Monotone sequence condition**

$$s_{\mathsf{resp}} > s_{\mathsf{req}}.$$

2. **Bounded time skew condition**

$$|t_{\mathsf{resp}} - t_{\mathsf{req}}| \le \Delta,$$

with Δ the allowed clock skew enforced by `mpdc_packet_time_valid`.
These guarantees prevent replay of older keep alive packets and ensure that each authenticated request corresponds to a current liveness event.

Exchange summary. The keep alive exchange is therefore:

$$X \xrightarrow{\texttt{keepalive\_request}} D \quad \text{and} \quad D \xrightarrow{\texttt{keepalive\_response}} X,$$

with both messages authenticating the sender identity and the time–sequence subheader. The resulting liveness records ensure that stale, offline, or impersonated nodes cannot maintain active status, and they enforce consistent time anchored freshness across the MPDC control plane.

3.2.7 Network Resignation Exchange

The network resignation exchange provides an authenticated mechanism for a device X to withdraw itself from the MPDC domain. The exchange ensures that the DLA learns of the departure in a cryptographically verifiable manner, updates its topology and certificate stores accordingly, and communicates back to the device that its resignation has been accepted. Both messages include signed time–sequence transcripts to prevent spoofing and replay.
The exchange uses the flags `resign_request` and `resign_response`, with sequence constants `NETWORK_RESIGN_REQUEST_SEQUENCE` and `NETWORK_RESIGN_RESPONSE_SEQUENCE`.

Let cert_X denote the device certificate, $\mathsf{ts} = (\mathsf{seq}, \mathsf{time})$ the request subheader, and let $\mathrm{Sign}_{\mathsf{sk}_X}$, $\mathrm{Sign}_{\mathsf{sk}_D}$ be signatures under device and DLA keys.

Resignation request. To initiate the exchange, the device constructs:

$$M_X = \mathsf{subhdr}(\mathsf{ts}) \parallel \mathsf{cert}_X \parallel \mathsf{intent},$$

where intent is a fixed byte sequence (or reserved field) indicating that the device requests removal from the topology. The request is authenticated by

$$\sigma_X = \mathrm{Sign}_{\mathsf{sk}_X}\big(H(M_X)\big).$$

The on wire message is

$$X \to D: \quad m_{\mathsf{res\_req}} = \big(\mathsf{flag},\ \mathsf{seq},\ \mathsf{time},\ \mathsf{cert}_X,\ \mathsf{intent},\ \sigma_X\big),$$

The DLA accepts a resignation request only under the following constraints:

1. **Certificate validity** under the RDS root.
2. **Topology membership:** $\mathsf{serial}(\mathsf{cert}_X)$ must index an active topology node.

3. **Freshness:** t_{req} must satisfy the MPDC time validity check

$$\texttt{mpdc\_packet\_time\_valid}(t_{\text{req}}) = \text{true}.$$

4. **Authenticated intent:**

$$\text{Verify}_{\text{pk}_X}(\sigma_X, H(M_X)) = \text{true}.$$

If any constraint fails, the DLA rejects the resignation and returns an error code instead of a response.

Topology state transition. If the request is accepted, the DLA removes the device from the active topology. Let $\text{id} = \text{serial}(\text{cert}_X)$. The update is:

$$\text{TopoStore}(D) \leftarrow \text{TopoStore}(D) \setminus \{\text{node}_D(\text{id})\},$$
$$\text{CertStore}(D) \leftarrow \text{CertStore}(D) \setminus \{\text{cert}_X\}.$$

The node is thereby marked inactive; future exchanges involving this identifier will fail topology lookups.

Resignation response. The DLA constructs a confirmation message with a fresh time–sequence $\text{ts}' = (s_{\text{resp}}, t_{\text{resp}})$:

$$M_D = \text{subhdr}(\text{ts}') \parallel \text{cert}_X \parallel \text{ack},$$

where ack is an acknowledgment field indicating successful removal. The DLA signs

$$\sigma_D = \text{Sign}_{\text{sk}_D}\big(H(M_D)\big).$$

The full response is

$$D \to X: \quad m_{\text{res\_resp}} = \big(\text{flag}, \text{seq}', \text{time}', \text{cert}_X, \text{ack}, \sigma_D\big),$$

Device X verifies the DLA signature and accepts the resignation acknowledgment. Upon doing so, it should clear its own session context, cached topology entries, and any outstanding MPDC-managed state.

Exchange summary. The resignation exchange is a short authenticated protocol:

$$X \xrightarrow{\texttt{resign\_request}} D, \qquad D \xrightarrow{\texttt{resign\_response}} X,$$

satisfying:
1. **Authenticated departure:** Requests are signed and bound to header time–sequence values. 2. **Topology removal:** The DLA updates its topology and certificate stores atomically. 3. **Replay protection:** Both parties enforce freshness via time, sequence monotonicity, and signature binding. 4. **State finalization:** A confirmed resignation deactivates the device globally within the MPDC control framework.
This exchange ensures that stale or disconnected devices cannot persist indefinitely in the topology and that the DLA, MAS, and Clients maintain a consistent and authenticated view of domain membership.

3.2.8 Certificate Revocation Broadcast and Response

The certificate revocation exchange provides a one-to-many authenticated mechanism for the DLA to invalidate a certificate across the entire MPDC domain. When the DLA determines that a certificate cert_Y belonging to some device Y must be revoked, it issues a broadcast revocation message that maps the certificate serial to a revocation status. All Clients, MAS instances, and Agents validate the signed broadcast, then remove the corresponding certificate and topology node from their local state. This process ensures that a revoked device cannot participate in any further MPDC exchanges.
Let serial_Y denote the revoked certificate serial, reason a fixed width revocation reason code, and $\mathsf{ts} = (\mathsf{seq}, \mathsf{time})$ the packet time–sequence fields. The revocation broadcast uses the flag `revoke_broadcast` and the fixed sequence constant `NETWORK_REVOKE_MESSAGE_SEQUENCE`.

Revocation broadcast construction. The DLA constructs the signed revocation transcript

$$M_D = \mathsf{subhdr}(\mathsf{ts}) \parallel \mathsf{serial}_Y \parallel \mathsf{reason},$$

binding the revocation data to the time–sequence fields. It computes

$$\sigma_D = \mathrm{Sign}_{\mathsf{sk}_D}\big(H(M_D)\big),$$

with `network_message_hash_sign` as in other DLA broadcast operations.
The broadcast packet is

$$D \to \{C, M, A_i\}_i : \quad m_{\mathsf{rev}} = \big(\mathsf{flag},\ \mathsf{seq},\ \mathsf{time},\ \mathsf{serial}_Y,\ \mathsf{reason},\ \sigma_D\big),$$

Revocation verification and response. All recipients use `network_revocation_message_verif` to validate the broadcast. Let X denote any recipient device. Upon receipt of m_{rev}, X:
1. Verifies header fields (flag, message length, sequence, and packet time validity). 2. Recomputes $\mathsf{subhdr}(\mathsf{ts})$ from the packet and reconstructs the transcript M_D. 3. Performs signature verification:

$$\mathsf{Verify}_{\mathsf{pk}_D}(\sigma_D, H(M_D)) = \mathsf{true}.$$

If the signature is valid, the device proceeds with the revocation update:

$$\mathsf{CertStore}(X) \leftarrow \mathsf{CertStore}(X) \setminus \{\mathsf{cert}_Y\},$$
$$\mathsf{TopoStore}(X) \leftarrow \mathsf{TopoStore}(X) \setminus \{\mathsf{node}_X(\mathsf{serial}_Y)\}.$$

This is implemented within `network_revoke_message_response`, which deletes the revoked certificate and removes the corresponding topology node entry based on its serial.

Mapping from serial to revocation status. The broadcast defines a signed mapping

$$\mathsf{serial}_Y \longmapsto \mathsf{revocation\_status}, \qquad \mathsf{revocation\_status} = \mathsf{reason},$$

authenticated by σ_D. Because the mapping is bound to $\mathsf{subhdr}(\mathsf{ts})$, replayed messages with mismatched sequence or stale timestamps are rejected. The revocation message is thus a cryptographically authenticated directive to remove a certificate and its associated node from all MPDC participants.

Effect on subsequent exchanges. After the revocation update:
1. Any certificate-based authentication involving cert_Y fails at signature verification or certificate lookup. 2. Topology-dependent routing or convergence functions treat Y as non-existent. 3. The DLA no longer includes the node in any future topology announcements.

Exchange summary. The certificate revocation mechanism is a broadcast–local update procedure:

$$D \xrightarrow{\texttt{revoke\_broadcast}} \{C, M, A_i\}, \qquad X \xrightarrow{\texttt{revocation\_response}} \text{(local state update)}.$$

Each recipient independently performs the removal, and no reply packet is sent back to the DLA. This ensures consistency of certificate and topology state across the full MPDC domain and prevents any revoked device from participating in subsequent authenticated communications.

3.3 Client Message Exchanges

3.3.1 Topology Query Exchange

The topology query exchange enables a Client C to obtain the network location, certificate, and topology metadata of a remote MPDC node Y. The Client sends a signed query to the DLA D containing its own certificate serial and an issuer string identifying the desired node. The DLA verifies the Client signature, looks up the topology node corresponding to the issuer string, and replies with an authenticated topology query response. The response allows the Client to synchronize its local certificate and topology caches with the authoritative DLA state.
Let serial_C denote the Client certificate serial, iss_Y the issuer string identifying the target node Y (for example, `"MAS"`, `"Agent:ID"`, or a DNS-like label), and $\mathsf{ts} = (\mathsf{seq}, \mathsf{time})$ the request time–sequence pair bound into the message. The exchange uses the flags `topology_query` and `topology_query_response`.

Topology query request. The Client constructs the transcript

$$M_C = \mathsf{subhdr}(\mathsf{ts}) \parallel \mathsf{serial}_C \parallel \mathsf{iss}_Y,$$

where $\mathsf{subhdr}(\mathsf{ts})$ is the fixed width serialization of the packet header fields. The Client signs the transcript hash:

$$\sigma_C = \mathrm{Sign}_{\mathsf{sk}_C}\big(H(M_C)\big).$$

The complete request is

$$\begin{aligned} C \to D : \quad m_{\mathsf{qreq}} = \big(&\mathsf{flag},\ \mathsf{seq},\ \mathsf{time},\ \mathsf{serial}_C,\\ &\mathsf{iss}_Y,\ \sigma_C\big), \end{aligned}$$

On receipt, the DLA performs:

1. Header validation (flag, time, sequence window).
2. Certificate lookup of the Client by serial_C:

   $$\mathsf{cert}_C = \mathsf{CertStore}(D)[\mathsf{serial}_C].$$

3. Verification of the signed transcript:

   $$\mathrm{Verify}_{\mathsf{pk}_C}(\sigma_C,\ H(M_C)) = \mathsf{true}.$$

4. Topology lookup:

   $$\mathsf{node}_D(Y) = \mathrm{ResolveIssuer}(\mathsf{iss}_Y),$$

   using the issuer string to locate the appropriate topology node, which contains cert_Y and the network address for Y.

Topology query response. Let $\mathsf{node}_D(Y)$ contain the serialized topology entry node_Y and its child certificate cert_Y. The DLA prepares a response with fresh subheader $\mathrm{ts}' = (s_{\mathsf{resp}}, t_{\mathsf{resp}})$ and constructs

$$M_D = \mathsf{subhdr}(\mathsf{ts}') \parallel \mathsf{cert}_Y \parallel \mathsf{node}_Y.$$

The DLA signs the transcript:

$$\sigma_D = \mathrm{Sign}_{\mathsf{sk}_D}\big(H(M_D)\big).$$

The full response is

$$D \to C : \quad m_{\mathsf{qresp}} = \big(\mathsf{flag},\ \mathsf{seq}',\ \mathsf{time}',\ \mathsf{cert}_Y,\ \mathsf{node}_Y,\ \sigma_D\big),$$

Client response verification and synchronization. Upon receiving the response, the Client performs:

1. Header validation (flag, message length, freshness).
2. Signature verification:
$$\mathrm{Verify}_{\mathsf{pk}_D}(\sigma_D,\ H(M_D)) = \mathsf{true}.$$
3. Certificate chain validation of cert_Y under the RDS root:
$$\texttt{mpdc\_network\_certificate\_verify}(\mathsf{cert}_Y) = \mathsf{true}.$$
4. Synchronization of local state:
$$\mathsf{CertStore}(C)[\mathsf{serial}_Y] \leftarrow \mathsf{cert}_Y,$$
$$\mathsf{TopoStore}(C)[Y] \leftarrow \mathsf{node}_Y.$$

The synchronized topology entry allows the Client to initiate secure connections with node Y, validate future messages from Y, and participate correctly in subsequent convergence or fragmentation key exchanges.

Exchange summary. The topology query exchange realizes the authenticated mapping

$$(\mathsf{serial}_C, \mathsf{iss}_Y) \longmapsto (\mathsf{cert}_Y, \mathsf{node}_Y),$$

bound through DLA and Client signatures. Its guarantees include:

1. **Authenticated query origin** through the Client signed transcript.

2. **Authoritative response** signed by the DLA.
3. **Certificate and topology synchronization** for the target node.
4. **Replay resistance** by binding time, sequence, and issuer string through signed hashing.

This exchange provides the foundational lookup mechanism that Clients use before establishing MPDC tunnel or fragmentation key exchanges with remote peers.

3.3.2 Topology Status Exchange

The topology status exchange gives a Client C a lightweight mechanism for querying the DLA D about the current synchronization state of a target node Y. Unlike the topology query exchange, which returns full certificate and topology records, the status exchange returns only a status code indicating whether the DLA has a valid and synchronized topology entry for Y. This allows Clients to decide whether additional synchronization steps, such as a topology query or incremental update, are required before attempting communication with the target node.
Let serial_C be the Client certificate serial, serial_Y the target node certificate serial, and $\mathsf{ts} = (\mathsf{seq}, \mathsf{time})$ the packet time–sequence pair. The exchange uses the flags `status_request` and `status_response`.

Topology status request. The Client constructs the signed transcript:

$$M_C = \mathsf{subhdr}(\mathsf{ts}) \parallel \mathsf{serial}_C \parallel \mathsf{serial}_Y,$$

where both serials are fixed width byte sequences. It signs the transcript using its device signing key:

$$\sigma_C = \mathrm{Sign}_{\mathsf{sk}_C}\big(H(M_C)\big).$$

The on wire request packet is:

$$C \to D : \quad m_{\mathsf{stat\_req}} = \big(\mathsf{flag},\ \mathsf{seq},\ \mathsf{time},\ \mathsf{serial}_C,\ \mathsf{serial}_Y,\ \sigma_C\big),$$

Upon receiving the request, the DLA:

1. Validates header and timestamp freshness.
2. Looks up the Client certificate using serial_C.
3. Verifies the signature:

$$\mathrm{Verify}_{\mathsf{pk}_C}(\sigma_C,\ H(M_C)) = \mathsf{true}.$$

4. Checks whether serial_Y indexes a known topology node:
$$\mathsf{node\_exists} = \big(\mathsf{serial}_Y \in \mathrm{Dom}(\mathsf{TopoStore}(D))\big).$$

The DLA then computes a status code for serial_Y.

Topology status codes. Let the status code be a single byte status with the following values:

$$\mathsf{status} = \begin{cases} \texttt{STATUS\_AVAILABLE}, & \text{if } \mathsf{node\_exists} = \mathsf{true} \text{ and entry is synchronized,} \\ \texttt{STATUS\_STALE}, & \text{if } \mathsf{node\_exists} = \mathsf{true} \text{ but requires reconciliation,} \\ \texttt{STATUS\_UNKNOWN}, & \text{otherwise.} \end{cases}$$

These codes reflect the underlying convergence state maintained by the DLA. In particular, a node is considered synchronized if it has been confirmed in a convergence or join update exchange within the acceptable freshness window.

Topology status response. The DLA constructs a response transcript:
$$M_D = \mathsf{subhdr}(\mathsf{ts}') \parallel \mathsf{serial}_Y \parallel \mathsf{status},$$
where $\mathsf{ts}' = (s_{\mathsf{resp}}, t_{\mathsf{resp}})$ is fresh. It signs the transcript:
$$\sigma_D = \mathrm{Sign}_{\mathsf{sk}_D}\big(H(M_D)\big).$$
The on wire response is:
$$D \to C: \quad m_{\mathsf{stat\_resp}} = \big(\mathsf{flag},\ \mathsf{seq}',\ \mathsf{time}',\ \mathsf{serial}_Y,\ \mathsf{status},\ \sigma_D\big),$$

Client response verification. The Client:
1. Validates header fields and freshness.
2. Verifies DLA signature:
$$\mathrm{Verify}_{\mathsf{pk}_D}(\sigma_D,\ H(M_D)) = \mathsf{true}.$$
3. Uses the returned status to decide whether:
    - the topology is ready for immediate use (`STATUS_AVAILABLE`);
    - the topology should be updated via a topology query or converge exchange (`STATUS_STALE`);
    - or the node is unknown and communication should not proceed (`STATUS_UNKNOWN`).

Exchange summary. The topology status exchange is the authenticated mapping:

$$(\mathsf{serial}_C, \mathsf{serial}_Y) \longmapsto \mathsf{status},$$

bound by signatures from both Client and DLA, with time and sequence freshness guarantees. The status codes reflect the underlying DLA convergence state, enabling Clients to make correct decisions about whether additional synchronization or certificate retrieval is necessary before initiating secure communications with the target node.

3.3.3 Fragmentation Key Exchange

The fragmentation key exchange establishes a symmetric fragmentation key k_{frag} shared between a Client C and the MAS M. This key is later used to authenticate and encrypt fragment collection messages, agent fragment queries, and other MAS coordinated operations. The exchange is a two message authenticated protocol. The Client is the requester and the MAS is the responder. Both sides bind the packet header time and sequence number into the signed transcript and into the AEAD associated data used for protecting key confirmation material.
Let cert_C be the Client certificate, node_C the Client topology node (serialized), and $\mathsf{ts} = (\mathsf{seq}, \mathsf{time})$ the time–sequence subheader of the request. The exchange uses flags `fragmentation_key_request` and `fragmentation_key_response`. The signed transcript format follows the general MPDC scheme. AEAD associated data is constructed from the packet header and subheader in the same fixed width encoding used elsewhere in the network code.

Fragmentation key request. The Client constructs the transcript

$$M_C = \mathsf{subhdr}(\mathsf{ts}) \parallel \mathsf{cert}_C \parallel \mathsf{node}_C,$$

and signs it:

$$\sigma_C = \mathrm{Sign}_{\mathsf{sk}_C}(H(M_C)).$$

The request packet is:

$$C \to M : \quad m_{\mathsf{fkreq}} = \big(\mathsf{flag},\ \mathsf{seq},\ \mathsf{time},\ \mathsf{cert}_C,\ \mathsf{node}_C,\ \sigma_C\big).$$

The MAS verifies the request with:

1. Header validation and time window.
2. Certificate lookup and chain validation under the RDS root.

3. Signature verification:

$$\mathrm{Verify}_{\mathsf{pk}_C}(\sigma_C,\ H(M_C)) = \mathsf{true}.$$

If all checks succeed, the MAS derives the fragmentation key:

$$k_{\mathsf{frag}} = \mathrm{KDF}\big(\mathsf{cert}_C,\ \mathsf{node}_C,\ \mathsf{ts}\big),$$

where KDF is instantiated as the MPDC approved cSHAKE based key derivation step implemented in the MAS code. The derivation binds Client identity, topology values, and the request freshness parameters.

Fragmentation key response. The MAS constructs the response transcript:

$$M_M = \mathsf{subhdr}(\mathsf{ts}') \parallel \mathsf{cert}_C \parallel k_{\mathsf{frag}},$$

where $\mathsf{ts}' = (s_{\mathsf{resp}}, t_{\mathsf{resp}})$ is a fresh time–sequence pair encoded in the response header. The MAS signs:

$$\sigma_M = \mathrm{Sign}_{\mathsf{sk}_M}(H(M_M)).$$

Before sending the response, the MAS runs:

$$C_{\mathsf{resp}} = \mathrm{AEAD\_Encrypt}\big(k_{\mathsf{frag}}, M_M,\ \mathsf{AD} = \mathsf{header}_{\mathsf{resp}}\big),$$

where $\mathsf{header}_{\mathsf{resp}}$ is the fixed width MPDC header concatenated with $\mathsf{subhdr}(\mathsf{ts}')$. These fields form the associated data for the AEAD transform, exactly as specified by the MPDC authenticated encryption profile.
The response packet is:

$$M \to C: \quad m_{\mathsf{fkresp}} = \big(\mathsf{flag} = \texttt{flag},\ \mathsf{seq}' = s_{\mathsf{resp}},\ \mathsf{time}' = t_{\mathsf{resp}},\ C_{\mathsf{resp}},\ \sigma_M\big).$$

Client verification and key confirmation. The Client:
1. Validates the header and time window. 2. Verifies the MAS signature:

$$\mathrm{Verify}_{\mathsf{pk}_M}(\sigma_M, H(M_M)) = \mathsf{true}.$$

3. Reconstructs the associated data from the response header exactly as the MAS produced it. 4. Decrypts:

$$M_M = \mathrm{AEAD\_Decrypt}(k_{\mathsf{frag}}, C_{\mathsf{resp}}, \mathsf{AD}),$$

which succeeds only if the fragmentation key is correct.
The presence of k_{frag} inside the response transcript, protected under the AEAD and bound to the MAS signature, provides explicit key confirmation. Both sides end with the same symmetric fragmentation key, ready for use in fragment collection and related MAS managed operations.

Exchange summary. The fragmentation key exchange implements a two message authenticated key establishment:

$$C \xrightarrow{\texttt{fragmentation\_key\_request}} M \quad \text{and} \quad M \xrightarrow{\texttt{fragmentation\_key\_response}} C,$$

with:
1. Explicit binding of Client identity and topology into the key derivation. 2. Use of the MPDC message header and subheader as AEAD associated data. 3. Explicit key confirmation enforced through AEAD decryption on the Client side. 4. Complete authentication through Client and MAS signatures.
This exchange establishes the fundamental symmetric key used to authenticate and encrypt subsequent fragmentation based operations within the MAS communication framework.

3.3.4 Fragment Collection Exchange

The fragment collection exchange enables a Client C to obtain a set of encrypted fragments from the MAS M. These fragments may originate from multiple Agents and are aggregated by the MAS under the fragmentation key previously established in the fragmentation key exchange. The exchange consists of a request from the Client and a response from the MAS. Both messages are authenticated and encrypted under the shared fragmentation key k_{frag}, with the MPDC packet header and subheader used as associated data in the AEAD transform.
Let $\mathsf{ts} = (\mathsf{seq}, \mathsf{time})$ denote the request packet time–sequence pair. The exchange uses flags `fragment_collection_request` and `fragment_collection_response`.

Fragment collection request. The Client prepares a list of fragment identifiers:

$$\mathsf{FID} = (f_1, f_2, \dots, f_m),$$

where each f_i is a fixed width identifier denoting a fragment required from the MAS aggregated store.
The request plaintext is:

$$P_C = \mathsf{subhdr}(\mathsf{ts}) \parallel \mathsf{FID}.$$

The Client encrypts the plaintext under the fragmentation key using:

$$C_{\mathsf{req}} = \text{AEAD\_Encrypt}\big(k_{\mathsf{frag}}, P_C,\ \mathsf{AD} = \mathsf{header}_{\mathsf{req}}\big),$$

where $\mathsf{header}_{\mathsf{req}}$ includes the MPDC packet header plus the serialized time–sequence subheader, exactly as constructed by `network_subheader_serialize`. The request sent on the wire is:

$$C \to M : \quad m_{\mathsf{fcreq}} = \big(\mathsf{flag},\ \mathsf{seq} = s_{\mathsf{req}},\ \mathsf{time} = t_{\mathsf{req}},\ C_{\mathsf{req}}\big).$$

Upon receiving the request, the MAS:
1. Validates time and sequence using `mpdc_packet_time_valid`. 2. Reconstructs the same associated data AD from the header. 3. Decrypts:

$$P_C = \text{AEAD\_Decrypt}(k_{\mathsf{frag}}, C_{\mathsf{req}}, \mathsf{AD}),$$

which fails if the request is malformed or if the fragmentation key is incorrect. 4. Parses the fragment identifiers FID and proceeds to collect fragments from its local store or by querying Agents.

MAS fragment aggregation. For each identifier f_i, the MAS retrieves the fragment:

$$\mathsf{Frag}(f_i) = \big(\mathsf{ctxt}_C(f_i),\ \mathsf{ctxt}_M(f_i)\big),$$

where: - $\mathsf{ctxt}_C(f_i)$ is the client encrypted version of the fragment, - $\mathsf{ctxt}_M(f_i)$ is the MAS encrypted version used for internal verification.
This structure is the result of prior Agent fragment query exchanges. The MAS assembles an ordered sequence of fragments:

$$\mathsf{FColl} = \big(\mathsf{Frag}(f_1),\ \dots,\ \mathsf{Frag}(f_m)\big).$$

Fragment collection response. The MAS constructs the plaintext response:

$$P_M = \mathsf{subhdr}(\mathsf{ts}') \parallel \mathsf{FColl},$$

where $\mathsf{ts}' = (s_{\mathsf{resp}}, t_{\mathsf{resp}})$ is fresh and FColl is encoded as the ordered list of fragment ciphertext pairs.
The MAS encrypts:

$$C_{\mathsf{resp}} = \text{AEAD\_Encrypt}\big(k_{\mathsf{frag}}, P_M,\ \mathsf{AD} = \mathsf{header}_{\mathsf{resp}}\big),$$

with associated data constructed from the MAS response header.
The response packet is:

$$M \to C : \quad m_{\mathsf{fcresp}} = \big(\mathsf{flag},\ \mathsf{seq}' = s_{\mathsf{resp}},\ \mathsf{time}' = t_{\mathsf{resp}},\ C_{\mathsf{resp}}\big).$$

Client verification and fragment assembly. The Client:
1. Validates the header fields. 2. Reconstructs the associated data. 3. Decrypts the response:

$$P_M = \text{AEAD\_Decrypt}(k_{\mathsf{frag}}, C_{\mathsf{resp}}, \mathsf{AD}).$$

4. Parses FColl into the ordered list of fragment ciphertext pairs. 5. Caches the client encrypted fragments $\mathsf{ctxt}_C(f_i)$ for later tunnel use or local processing.
Because the MAS includes a fresh subheader in the response and because AEAD binds the header to the ciphertext, the Client receives both authenticity and freshness guarantees for the fragments.

Exchange summary. The fragment collection exchange realizes the mapping:

$$(f_1, \dots, f_m) \longmapsto \big(\mathsf{Frag}(f_1),\ \dots,\ \mathsf{Frag}(f_m)\big),$$

protected under k_{frag}. Its properties include:
1. **Ordered fragment semantics** through preservation of FID ordering. 2. **Authenticated encryption** using the fragmentation key and MPDC header associated data. 3. **Integrity of fragment origins** through MAS mediated Agent fragment queries. 4. **Replay protection** enforced by time–sequence freshness and AEAD binding.
This exchange is the primary mechanism through which Clients obtain the fragment material needed to reconstruct keys, tunnel credentials, or agent supplied data within the MPDC framework.

3.4 Agent Message Exchanges

3.4.1 Agent Fragment Query Exchange

The Agent fragment query exchange allows the MAS M to request a fresh fragment from an Agent A. Each fragment is generated randomly by the Agent and encrypted under two master fragmentation keys: the MAS master key k^M_{MFK} and the Client master key k^C_{MFK}. The Agent returns both ciphertexts and a MAC computed over their concatenation using a MAC key derived from these master keys and a fresh token. The MAS verifies the MAC and stores the resulting fragment pair for subsequent fragment collection responses.
Let $\mathsf{ts} = (\mathsf{seq}, \mathsf{time})$ be the time–sequence subheader for the query request. The exchange uses flags `fragment_query_request` and `fragment_query_response`.

Fragment query request. The MAS sends a minimal request containing the subheader serialized as associated data for the upcoming MAC verification:

$$M \to A : \quad m_{\mathsf{fqreq}} = \big(\mathsf{flag},\ \mathsf{seq} = s_{\mathsf{req}},\ \mathsf{time} = t_{\mathsf{req}}\big).$$

The Agent uses the subheader as part of the transcript for the MAC computation, providing freshness guarantees at the MAC level.

Agent fragment generation and encryption. Upon receiving the request, the Agent generates a fresh random fragment:

$$F \overset{\$}{\leftarrow} \{0,1\}^{\ell_{\mathsf{frag}}}.$$

Let k^{M}_{MFK} be the MAS master fragmentation key, and k^{C}_{MFK} the Client master fragmentation key, both derived earlier from the MFK exchange. The Agent encrypts the fragment twice:

$$C_M = \text{AEAD\_Encrypt}\big(k^{M}_{\mathsf{MFK}},\ F,\ \mathsf{AD}_M\big),$$
$$C_C = \text{AEAD\_Encrypt}\big(k^{C}_{\mathsf{MFK}},\ F,\ \mathsf{AD}_C\big),$$

where AD_M and AD_C are associated data values constructed from the MAS and Client identities and the serialized MAS subheader. Both ciphertexts include their authentication tags.

MAC computation. The Agent derives a MAC key:

$$k_{\mathsf{MAC}} = \text{KDF}\big(k^{M}_{\mathsf{MFK}} \parallel k^{C}_{\mathsf{MFK}} \parallel \mathsf{token}\big),$$

where token is a freshly generated random nonce used to prevent key reuse. The Agent computes:

$$T = \text{MAC}_{k_{\mathsf{MAC}}}(C_M \parallel C_C).$$

Fragment query response. The response sent to the MAS is:

$$A \to M : \quad m_{\mathsf{fqresp}} = \big(\mathsf{flag},\ \mathsf{seq}' = s_{\mathsf{resp}},\ \mathsf{time}' = t_{\mathsf{resp}},\ C_M,\ C_C,\ \mathsf{token},\ T\big).$$

The response body consists of the ciphertext pair (C_M, C_C), the fresh token, and the MAC value.

MAS verification. Upon receiving the response, the MAS:
1. Validates header freshness. 2. Reconstructs the MAC key:

$$k_{\mathsf{MAC}} = \text{KDF}\big(k^{M}_{\mathsf{MFK}} \parallel k^{C}_{\mathsf{MFK}} \parallel \mathsf{token}\big).$$

3. Computes:

$$T' = \text{MAC}_{k_{\mathsf{MAC}}}(C_M \parallel C_C).$$

4. Accepts the response only if:

$$T' = T.$$

If the MAC is valid, the MAS stores:

$$\mathsf{Frag}(f_{\text{new}}) = (C_C, C_M),$$

indexed under a fresh fragment identifier f_{new}.

Exchange summary. The Agent fragment query exchange implements:

$$M \xrightarrow{\texttt{fragment\_query\_request}} A \quad \text{and} \quad A \xrightarrow{\texttt{fragment\_query\_response}} M,$$

with the following properties:

1. **Fresh fragment generation:** Each fragment is unpredictable and unique.
2. **Dual encryption:** One ciphertext for MAS use, one for Client use.
3. **Derived MAC key:** The MAC binds both ciphertexts together and certifies that they correspond to the same underlying fragment.
4. **Strong verification:** MAS accepts a fragment only if the MAC matches exactly under the derived key.
5. **Replay resistance:** Use of the serialized time–sequence header and a fresh token prevents reuse of prior fragments.

This exchange ensures that both MAS and Client later decrypt the same underlying fragment while allowing the MAS to verify fragment correctness without needing to decrypt it."‘

3.4.2 Master Fragment Key Exchange

The master fragment key (MFK) exchange is a four message authenticated key establishment protocol between the MAS M and an Agent A. The exchange resembles an authenticated KEM with explicit key confirmation. The result is a shared master fragmentation key

$$k_{\mathsf{MFK}} \in \{0,1\}^{\ell_{\mathsf{mfk}}},$$

which is used to encrypt fragments and derive downstream MAC and AEAD keys. The protocol is realized by the functions in `network.c`:
`network_mfk_request_packet`, `network_mfk_establish_packet`,

`network_mfk_response_packet`, `network_mfk_verify_packet`, and the public orchestrator `mpdc_network_mfk_exchange_response`.
We denote the MAS certificate by cert_M, the Agent certificate by cert_A, and the request and response subheaders by $\mathsf{subhdr(ts)}$ and $\mathsf{subhdr(ts')}$.
The exchange consists of four authenticated messages:

MAS ⍰ Agent: MFK Request. MAS constructs the transcript

$$M_M^{\mathsf{req}} = \mathsf{subhdr(ts)} \parallel \mathsf{cert}_M,$$

and signs it:

$$\sigma_M^{\mathsf{req}} = \mathrm{Sign}_{\mathsf{sk}_M}\big(H(M_M^{\mathsf{req}})\big).$$

The message is:

$$M \to A : \quad m_{\mathsf{mfk\_req}} = \big(\mathsf{flag},\ \mathsf{ts},\ \mathsf{cert}_M,\ \sigma_M^{\mathsf{req}}\big).$$

Agent verifies:
1. $\mathrm{Verify}_{\mathsf{pk}_M}(\sigma_M^{\mathsf{req}}, H(M_M^{\mathsf{req}})) = \mathsf{true}$; 2. cert_M is valid under the RDS root; 3. Header time–sequence freshness.

Agent ⍰ MAS: MFK Establish Agent generates an asymmetric key pair:

$$(sk_A^{\mathsf{kem}},\ pk_A^{\mathsf{kem}}) \leftarrow \mathrm{KEM.KeyGen}().$$

It constructs:

$$M_A^{\mathsf{est}} = \mathsf{subhdr(ts')} \parallel \mathsf{cert}_A \parallel pk_A^{\mathsf{kem}},$$

and signs:

$$\sigma_A^{\mathsf{est}} = \mathrm{Sign}_{\mathsf{sk}_A}\big(H(M_A^{\mathsf{est}})\big).$$

The establish message is:

$$A \to M : \quad m_{\mathsf{mfk\_est}} = \big(\mathsf{flag},\ \mathsf{ts}',\ \mathsf{cert}_A,\ pk_A^{\mathsf{kem}},\ \sigma_A^{\mathsf{est}}\big).$$

MAS verifies:
1. $\mathrm{Verify}_{\mathsf{pk}_A}(\sigma_A^{\mathsf{est}}, H(M_A^{\mathsf{est}}))$; 2. cert_A under RDS root; 3. Freshness of ts'.

MAS ⍰ Agent: MFK Response (KEM Encapsulation) MAS samples a shared secret:

$$k_{\mathsf{MFK}} \xleftarrow{\$} \{0,1\}^{\ell_{\mathsf{mfk}}}.$$

It encapsulates the key under the Agent's public key:

$$C_{\mathsf{kem}} = \text{KEM.Encaps}(pk_A^{\mathsf{kem}},\ k_{\mathsf{MFK}}).$$

It constructs:

$$M_M^{\mathsf{resp}} = \mathsf{subhdr}(\mathsf{ts}'') \parallel C_{\mathsf{kem}},$$

and signs:

$$\sigma_M^{\mathsf{resp}} = \text{Sign}_{\mathsf{sk}_M}\big(H(M_M^{\mathsf{resp}})\big).$$

The response message is:

$$M \to A: \quad m_{\mathsf{mfk\_resp}} = \big(\mathsf{flag},\ \mathsf{ts}'',\ C_{\mathsf{kem}},\ \sigma_M^{\mathsf{resp}}\big).$$

Agent verifies MAS signature and then performs:

$$k'_{\mathsf{MFK}} = \text{KEM.Decaps}(sk_A^{\mathsf{kem}},\ C_{\mathsf{kem}}).$$

If the KEM is correct, then:

$$k'_{\mathsf{MFK}} = k_{\mathsf{MFK}}.$$

Agent ⍰ MAS: MFK Verify (Explicit Key Confirmation Agent constructs:

$$M_A^{\mathsf{ver}} = \mathsf{subhdr}(\mathsf{ts}''') \parallel H(k'_{\mathsf{MFK}}),$$

and signs:

$$\sigma_A^{\mathsf{ver}} = \text{Sign}_{\mathsf{sk}_A}\big(H(M_A^{\mathsf{ver}})\big).$$

The verify message is:

$$A \to M: \quad m_{\mathsf{mfk\_ver}} = \big(\mathsf{flag} = \texttt{mfk\_verify},\ \mathsf{ts}''',\ H(k'_{\mathsf{MFK}}),\ \sigma_A^{\mathsf{ver}}\big).$$

MAS verifies:
1. Signature under pk_A; 2. That $H(k'_{\mathsf{MFK}}) = H(k_{\mathsf{MFK}})$.
If the check succeeds, both parties now hold the same master fragmentation key.

Exchange summary. The complete sequence is:

$$
\begin{aligned}
M &\xrightarrow{\texttt{mfk\_request}} A, \\
A &\xrightarrow{\texttt{mfk\_establish}} M, \\
M &\xrightarrow{\texttt{mfk\_response}} A, \\
A &\xrightarrow{\texttt{mfk\_verify}} M.
\end{aligned}
$$

Security properties:

1. **Mutual authentication** via MAS and Agent certificates and signatures.
2. **KEM based key derivation** with encapsulation and decapsulation.
3. **Explicit key confirmation** through the final message.
4. **Freshness** enforced by time–sequence subheaders bound into every signed transcript.
5. **Replay protection** through unique nonces and serialized header fields.
6. **Forward secrecy** to the extent provided by the underlying KEM.

This establishes k_{MFK}, which becomes the basis for all subsequent fragment query and fragment collection operations.

3.5 MAS Message Exchanges

3.5.1 Tunnel Session Establishment and Encrypted Data Transfer

After the fragmentation keys and master fragment keys have been established, the MAS M and Client C derive tunnel traffic keys used to protect the encrypted channel between them. The tunnel layer uses a symmetric AEAD construction with explicit associated data incorporating the MPDC packet header, the time–sequence subheader, and tunnel specific counters. Tunnel establishment confirms that both parties hold matching send and receive keys, after which all tunnel messages carry encrypted application payloads with authenticated metadata. Additional messages provide transfer control and orderly shutdown.

Let the derived traffic keys be:

$$
k_{\mathsf{tx}}^{C\to M}, \qquad k_{\mathsf{rx}}^{M\to C},
$$

for Client to MAS and MAS to Client traffic, respectively. These keys are deterministic outputs of the MAS and Client key derivation functions using the same inputs: fragmentation keys, handshake transcripts, and topology fields. The MPDC header plus subheader is always used as AEAD associated data.

Tunnel Establishment Message. The initiator (usually the Client) assembles a tunnel establishment transcript:

$$M^{\text{est}} = \mathsf{subhdr}(\mathsf{ts}) \parallel H(k_{\text{tx}}^{C\to M}) \parallel H(k_{\text{rx}}^{M\to C}),$$

binding the hashed tunnel keys to the time–sequence fields.
The sender encrypts:

$$C_{\text{est}} = \text{AEAD\_Encrypt}\big(k_{\text{tx}}^{C\to M},\ M^{\text{est}},\ \mathsf{AD} = \mathsf{header}_{\text{est}}\big),$$

and transmits:

$$C \to M: \quad m_{\text{t\_est}} = \big(\mathsf{flag},\ \mathsf{ts},\ C_{\text{est}}\big).$$

Verification: The MAS reconstructs AD, decrypts with $k_{\text{rx}}^{M\to C}$, and checks that the hashes of its own derived keys match the received values. If so:

$$\text{tunnel\_state} \leftarrow \texttt{ESTABLISHED}.$$

This message therefore provides *explicit key confirmation* at the tunnel layer.

Encrypted Tunnel Data Messages Once the tunnel is established, all application data is protected under the AEAD transform:

$$C_{\text{data}} = \text{AEAD\_Encrypt}\big(k_{\text{tx}}^{C\to M},\ P,\ \mathsf{AD} = \mathsf{header}_{\text{data}}\big),$$

where P is the plaintext payload and

$$\mathsf{header}_{\text{data}} = \text{MPDC header} \parallel \mathsf{subhdr}(\mathsf{ts}) \parallel \texttt{tunnel\_counter}.$$

The tunnel counter is a monotonically increasing per direction value maintained in the session state, guaranteeing nonce uniqueness for AEAD. The MAS uses k_{rx} to decrypt Client transmissions, while the Client uses k_{rx} to decrypt MAS transmissions.
Messages have the form:

$$C \leftrightarrow M: \quad m_{\text{data}} = \big(\mathsf{flag} = \texttt{tunnel\_data},\ \mathsf{ts},\ \mathsf{counter},\ C_{\text{data}}\big).$$

Verification rule:

$$\text{AEAD\_Decrypt}(k_{\mathsf{rx}}, C_{\mathsf{data}}, \mathsf{AD}) \text{ must succeed.}$$

If decryption fails the packet is rejected.

Tunnel Transfer Request To transfer the tunnel to another MAS instance or transition to a new fragmentation key, the Client sends a transfer request:

$$C \to M : \quad m_{\mathsf{t\_xfer}} = \big(\mathsf{flag},\ \mathsf{ts},\ C_{\mathsf{xfer}}\big),$$

with:

$$C_{\mathsf{xfer}} = \text{AEAD\_Encrypt}(k_{\mathsf{tx}}^{C\to M}, P_{\mathsf{xfer}}, \mathsf{AD}).$$

The MAS processes the request, updates session parameters, and responds with the appropriate control message. Transfer messages are authenticated identically to ordinary tunnel data messages.

Tunnel Termination A termination message allows either side to cleanly close the tunnel:

$$X \to Y : \quad m_{\mathsf{t\_term}} = \big(\mathsf{flag},\ \mathsf{ts},\ C_{\mathsf{term}}\big).$$

Again:

$$C_{\mathsf{term}} = \text{AEAD\_Encrypt}(k_{\mathsf{tx}}, P_{\mathsf{term}}, \mathsf{AD}).$$

Upon successfully decrypting a termination packet, the receiver clears the tunnel state, zeros session keys, and transitions the session state machine to `CLOSED`.

Exchange Summary The tunnel exchange realizes a stateful, replay resistant encrypted channel:

$$\begin{aligned}
&\text{Key confirmation: } C \xrightarrow{\texttt{tunnel\_establish}} M,\\
&\text{Encrypted transport: } C \leftrightarrow M\ (\texttt{tunnel\_data}),\\
&\text{Control: } C \xrightarrow{\texttt{tunnel\_transfer\_request}} M,\\
&\text{Shutdown: } X \xrightarrow{\texttt{tunnel\_termination}} Y.
\end{aligned}$$

Security derives from:
1. **AEAD with unique per packet nonces** via tunnel counters. 2. **Associated data binding** of MPDC header, subheader, and counters. 3. **Explicit

key confirmation** in tunnel establishment. 4. **Deterministic, synchronized traffic key derivation** from fragmentation keys. 5. **Time–sequence freshness checks** at the message layer.
This defines the MAS–Client encrypted communication model on which all higher level MPDC functionality relies.

3.5.2 Connection Termination and Error Reporting

The connection termination and error reporting messages provide reliable shutdown and diagnostic signaling for all MAS coordinated exchanges. Termination messages allow either side, MAS M or Client C, to explicitly close an MPDC association. Error packets allow authenticated communication of failure conditions such as packet format violations, signature failures, time window violations, state machine errors, and cryptographic verification failures. Each message includes a time–sequence subheader that is bound into its authenticated transcript, preventing replay or injection of forged control signals. These control plane messages use the flags:
- `connection_terminate`, - `error_condition`, - `system_error`,

Connection Termination When either MAS or Client wishes to shut down a connection or tunnel, it constructs a termination transcript:

$$M^{\mathsf{term}} = \mathsf{subhdr(ts)} \parallel \mathsf{reason},$$

where reason is a fixed width enumerated termination code. The sender signs:

$$\sigma = \mathrm{Sign}_{\mathsf{sk}}(H(M^{\mathsf{term}})).$$

The message is:

$$X \to Y : \quad m_{\mathsf{term}} = \big(\mathsf{flag} = \texttt{connection\_terminate},\ \mathsf{ts},\ \mathsf{reason},\ \sigma\big).$$

The receiver verifies the signature using the corresponding certificate anchored in the RDS root and, upon acceptance, transitions its local state machine to `CLOSED`. This applies both to MAS–Client tunnels and to MAS–Agent exchanges.

Error Reporting During message parsing or cryptographic verification, either endpoint may detect a failure condition. The MPDC code exposes two types of error packets:
1. `error_condition`, signaling protocol level issues such as malformed packets, bad sizes, invalid flags, unexpected sequence numbers, replayed timestamps, or topology lookup failures.

2. `system_error`, signaling internal errors such as memory faults, key derivation failures, or cryptographic library failures.
Let ecode be a fixed width error code selected from the MPDC error code enumeration. The error message transcript is:

$$M^{\mathsf{err}} = \mathsf{subhdr}(\mathsf{ts}) \parallel \mathsf{ecode},$$

with signature:

$$\sigma = \mathrm{Sign}_{\mathsf{sk}}(H(M^{\mathsf{err}})).$$

The message on the wire is:

$$X \to Y : \quad m_{\mathsf{err}} = \big(\mathsf{flag} = \texttt{error\_condition} \text{ or } \texttt{system\_error},\ \mathsf{ts},\ \mathsf{ecode},\ \sigma\big).$$

Integration with State Machines Error and termination packets fit into the MAS and Client state machines as follows:

During handshake protocols (fragmentation key, fragment query, MFK exchange): A received error immediately transitions the session to `FAILED` state. A termination transitions the exchange to `CLOSED`. Retries are not permitted under MFK or fragment query errors; state is rolled back.

During a tunnel session: A `tunnel_data` packet expected after `tunnel_establish` but followed by an `error_condition` forces shutdown of the tunnel. A termination message must be acknowledged by cleaning session keys and deallocating counters.

For DLA related exchanges: Errors originating from MAS or Client in DLA directed messages (keep alive, join update, topology requests) cause the DLA to mark the node as unsynchronized or unhealthy, prompting further convergence or removal operations.

Security Properties The security guarantees provided by these control messages stem from:

1. **Signed time–sequence binding** that prevents replay or out of order injection of termination or error signals.
2. **Authenticated origin** via signatures under device or MAS keys.

3. **Well defined state transitions** preventing protocol downgrade or desynchronization attacks.
4. **Isolation from AEAD transforms**: error and termination messages do not alter key material or counters, preserving AEAD nonce safety.

Exchange Summary The control message flow is:

$$X \xrightarrow{\texttt{connection\_terminate}} Y, \qquad X \xrightarrow{\texttt{error\_condition/system\_error}} Y.$$

These messages maintain orderly shutdown, communicate failures clearly, and preserve message layer security without weakening the invariants of the various MAS mediated exchanges.

4 Execution Model and Security Definitions

This section defines the execution environment used to model MPDC-I, the capabilities of the adversary, the structure of tunnel sessions, the corruption model, and the security experiments for authenticated key exchange, ciphertext integrity, replay resistance, and post compromise recovery. The model reflects the engineering description in Section 2 and provides the formal basis for the proofs in later sections.

4.1 System and Adversarial Model

The system consists of a Client $\mathcal{C}$, an MPDC Access Server $\mathcal{S}$, and a set of Agents $\mathcal{A} = \{A_1, \ldots, A_n\}$. Each device possesses a long term signing key and a certificate issued under the RDS root. All devices validate signatures under the RDS key. The DLA acts as a trusted management service that distributes topology and certificate updates but does not participate directly in fragment collection or tunnel establishment.

A protocol execution is composed of one or more sessions. A session is a complete run of the fragment collection and tunnel establishment process between $\mathcal{C}$ and $\mathcal{S}$ using the set of Agents that are active at the time of the run. Each device may maintain multiple sessions across time but no device participates in more than one active session at once.

The adversary controls the network. It may schedule all message deliveries, reorder or delay messages, inject or modify messages, and drop messages. All communication takes place over channels controlled by the adversary, subject to signature and MAC verification rules derived from certificate validation and symmetric tunnel authentication. The adversary may compromise devices according to the corruption oracles defined below. The adversary does not learn the internal random coins of honest parties except through corruption oracles. The adversary is considered efficient if it runs in polynomial time in the security parameter. All advantage functions are defined with respect to efficient adversaries.

4.2 MPDC Session Identifiers and Transcript Structure

A session is identified by the tuple

$$\mathsf{sid} = (\mathcal{C}, \mathcal{S}, \mathcal{A}_{\text{active}}, \tau)$$

where $\mathcal{A}_{\text{active}} \subseteq \mathcal{A}$ is the ordered set of Agents that participate in fragment collection and τ is the MAS token used for that round.

The session transcript consists of:

- the authenticated asymmetric exchanges that established the long term Master Fragment Keys between each pair of devices that share an MFK,
- the MAC protected and MFK derived fragment deliveries from each active Agent,
- the MAS fragment,
- the aggregation inputs used by $\mathcal{C}$ and $\mathcal{S}$,
- the encrypted tunnel messages that follow session establishment.

The session key SK is defined as the pair of transmit and receive keys derived from the aggregate value h produced by the fragment reconstruction process. Both parties attach the same SK to the unique session identifier determined by the fragment round.

4.3 Agent Corruption and Trust Anchor Oracles

The following oracles formalize how the adversary may compromise entities in the system.

Definition 4.1 (Agent Corruption Oracle). *The oracle* $\mathsf{CorruptAgent}(A_i)$ *returns the long term signing key of* A_i*, the certificate of* A_i*, and all Master Fragment Keys that* A_i *shares with honest peers. The oracle does not reveal any ephemeral fragment keys or any tunnel keys derived from fragments supplied by honest Agents.*

Definition 4.2 (Client and MAS Corruption Oracles). *The oracle* CorruptClient() *returns all long term secrets held by $\mathcal{C}$, including its signing key and all MFK values. The oracle* CorruptMAS() *returns the corresponding secrets for $\mathcal{S}$. These oracles do not reveal tunnel keys from sessions that have not yet completed.*

Definition 4.3 (DLA Corruption Oracle). *The oracle* CorruptDLA() *gives the adversary full control of the DLA signing key. After corruption the adversary may forge topology, convergence, and revocation messages. The oracle does not reveal the RDS root key.*

The adversary may query these oracles at any time except for restrictions imposed by the admissibility rules of the security experiments.

4.4 The MPDC Indistinguishability Experiment

Authenticated key exchange security for MPDC-Is defined through the experiment $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{MPDC}}$, which we now describe.

Definition 4.4 (Experiment $\mathsf{Exp}^{\mathcal{A}}_{\mathsf{MPDC}}$). *The experiment proceeds as follows.*

Setup. *The challenger generates the long term keys and certificates for RDS, DLA, $\mathcal{C}$, $\mathcal{S}$, and all Agents. The challenger samples the topology, the configuration set, and initializes all local state. The adversary receives the public parameters, including all certificates.*

Adversarial Interface. *The adversary interacts with the system through the following operations.*

- ***Send**: deliver a message to any device instance.*
- ***Receive**: observe outgoing messages.*
- ***Activate**: trigger a device to start a session or execute a protocol step.*
- ***Corrupt**: query any corruption oracle that is permitted by the rules of the experiment.*

Challenge. *The adversary designates a target session, subject to admissibility conditions. The challenger samples a bit b. If $b = 0$, the challenger provides the real session key* SK. *If $b = 1$, the challenger provides a uniformly random key of the same length. The adversary continues to interact with the system but may not corrupt any entity that would violate admissibility.*

Output. *The adversary outputs a guess b'. The experiment outputs* 1 *if $b' = b$ and* 0 *otherwise.*

Definition 4.5 (MPDC AKE Security). *The advantage of an adversary $\mathcal{A}$ in the MPDC authenticated key exchange experiment is*

$$\mathsf{Adv}^{\mathsf{AKE}}_{\mathsf{MPDC}}(\mathcal{A}) = \left|\Pr[\mathsf{Exp}^{\mathcal{A}}_{\mathsf{MPDC}} = 1] - \tfrac{1}{2}\right|.$$

MPDC-I is AKE secure if this advantage is negligible.

4.5 Integrity and Replay Security Games

The tunnel uses authenticated encryption under RCS or AES-GCM with associated data that includes the sequence number and timestamp. Integrity and replay resistance are modeled through two experiments.

Definition 4.6 (Ciphertext Integrity Game). *An adversary wins if it produces a ciphertext and tag that an honest party accepts in a session for which the adversary never provided the corresponding plaintext to the encryption oracle. The advantage in this game is denoted $\mathsf{Adv}^{\mathsf{INT}}_{\mathsf{MPDC}}(\mathcal{A})$.*

Definition 4.7 (Replay Game). *An adversary wins if it causes an honest party to accept the same message twice within the same session. Since sequence numbers and timestamps appear in the associated data, the only possible success is to forge a valid tag on a modified header. The advantage in this game is denoted $\mathsf{Adv}^{\mathsf{Replay}}_{\mathsf{MPDC}}(\mathcal{A})$.*

4.6 Post Compromise Security Game

Post compromise security models recovery from compromise of $\mathcal{C}$ or $\mathcal{S}$.

Definition 4.8 (PCS Game). *An adversary compromises $\mathcal{C}$ or $\mathcal{S}$ at time t and learns all secrets known at that time. At a later time $t + 1$ a new fragment collection round is executed with at least one honest Agent. The adversary receives either the real key for the round at time $t + 1$ or a random key and must guess which value it received. The advantage in this game is denoted $\mathsf{Adv}^{\mathsf{PCS}}_{\mathsf{MPDC}}(\mathcal{A})$.*

The protocol provides PCS if $\mathsf{Adv}^{\mathsf{PCS}}_{\mathsf{MPDC}}(\mathcal{A})$ is negligible for all efficient adversaries.

5 Entropy Contribution of Honest Agents

This section formalizes the randomness contributed by Agents, proves that one honest Agent suffices to guarantee full entropy of the aggregate value, and analyzes collusion scenarios in which all other Agents behave adversarially. The goal is to justify the central security claim that the MPDC tunnel key remains indistinguishable from random as long as at least one Agent provides an uncompromised fragment.

5.1 Randomness Model for Fragments

Each Agent A_i samples a 256 bit fragment F_i using the system random number generator implemented through SHAKE based expansion. For honest Agents the fragment distribution is modeled as a uniform random variable over $\{0, 1\}^{256}$. Malicious Agents may choose their fragments adversarially and may adapt their choices based on the public transcript and on all information available through corruption oracles. The adversary does not observe any internal randomness used by an honest Agent except through the masked fragment transmitted via the protected fragment channel.

For each Agent A_i, the MAS token τ, and the certificate hashes (h_i, h_{peer}), the Agent derives an Ephemeral Fragment Key efk_i using the MFK based Keccak derivation routine implemented in the protocol. The masked fragment is computed as

$$C_i = F_i \oplus \mathsf{SHAKE}(\mathsf{efk}_i, 256),$$

and the Agent returns C_i along with a KMAC tag computed under efk_i and the timestamp and sequence values serialized in the network sub header. The tag prevents modification of C_i or the header. Fragment replies are authenticated solely with a KMAC tag under efk_i; no additional signature is applied at the fragment layer.

Upon receiving valid fragments from the active set of Agents, the Client and the MAS reconstruct the unmasked fragments and absorb them into a Keccak based state in the deterministic order determined by the topology list. Let

$$h = \mathsf{SHAKE}(F_{\text{MAS}} \parallel F_1 \parallel \cdots \parallel F_k),$$

where the concatenation follows the sorted list of active Agents and F_{MAS} is the MAS fragment. The value h is the input to the tunnel key derivation function.

The adversary may control all F_i for corrupted Agents and may observe the masked versions of all F_i for honest Agents but cannot recover the honest fragments unless it forges the KMAC tag or derives efk_i by breaking the MFK based derivation function.

5.2 Entropy Lemma for One Honest Agent

Lemma 5.1 (Entropy from One Honest Agent). *Let F^* be the fragment of an honest Agent, sampled uniformly from $\{0,1\}^{256}$. Let $F_1, \dots, F_m$ be adversarially chosen fragments that may depend on the public transcript and on all corruption oracles except the honest Agent. Let*

$$h = \mathsf{SHAKE}(F_{MAS} \,\|\, F_1 \,\|\, \cdots \,\|\, F_m \,\|\, F^*)$$

be the aggregate value derived from the Keccak based extraction process. In the random oracle model for SHAKE *or under the indifferentiability of Keccak from a random oracle, the value h has min entropy at least $256 - \varepsilon$ conditioned on the adversary view, where ε is negligible in the security parameter.*

Proof. The proof follows from standard extraction results for Keccak based random oracles. Since F^* is sampled uniformly from $\{0,1\}^{256}$ and remains hidden from the adversary, the concatenation:

$$X = F_{\mathrm{MAS}} \,\|\, F_1 \,\|\, \cdots \,\|\, F_m \,\|\, F^*$$

contains a block of 256 bits that is uniform and independent of all adversarially generated components.

Model SHAKE as a random oracle H that maps arbitrary strings to arbitrary length outputs. For any fixed prefix $P = F_{\mathrm{MAS}} \,\|\, F_1 \,\|\, \cdots \,\|\, F_m$, the value $h = H(P\|F^*)$ is distributed uniformly over the output space conditioned only on the input string. Since F^* is uniform and never revealed to the adversary, the input to H ranges over a space of size 2^{256} and is independent of all adversary controlled values.

If the adversary does not query H on the exact value $P\|F^*$, then h is uniform from the adversary's perspective. The probability that the adversary submits this exact query is at most $q_h/2^{256}$ where q_h is the number of oracle queries, which is negligible.

Formally, if U is a uniformly distributed output of the same length as h, then:

$$|\Pr[h = y] - \Pr[U = y]| \le \frac{q_h}{2^{256}}.$$

It follows that h has min entropy at least $256 - \log_2(q_h)$ which is $256 - \varepsilon$ for negligible ε.

Therefore one honest fragment suffices to ensure that the aggregate hash remains indistinguishable from random under the assumed properties of Keccak. □

5.3 Threshold View and Collusion Scenarios

The fragment system can be viewed as a one of n threshold construction. The adversary may corrupt up to $n-1$ Agents and may collude with the MAS or the Client if those entities are corrupted. Corrupted Agents learn their own fragments and may attempt to infer the honest fragment from the masked version. However the KMAC tag protects the masked fragment, and the ephemeral fragment key remains unknown to the adversary unless the corresponding MFK or signing key is compromised.

Suppose $n - 1$ Agents are corrupted and the MAS is also compromised. The adversary then learns every fragment except F^*. Since F^* remains uniformly random, the aggregate input to the KDF still contains a full entropy component. The lemma above shows that the output h retains full entropy from the adversary's point of view. Consequently the derived tunnel keys remain indistinguishable from random.

This threshold property is central to the security of MPDC. It ensures that compromise of the MAS and all but one of the Agents does not reveal the tunnel key. A symmetric argument shows that compromise of the Client and all but one Agent also leaves the tunnel key unpredictable. The fragment system therefore provides robustness against partial compromise and supports recovery of security during new fragment collection rounds.

6 Provable Security Analysis

This section states the assumptions on the underlying primitives and provides reduction based proofs for the confidentiality of Master Fragment Keys, the indistinguishability of the tunnel keys, and the integrity and replay resistance of the encrypted tunnel. The analysis makes use of the execution model and entropy lemma presented in the earlier sections.

6.1 Assumptions on the Underlying Primitives

Security of MPDC-I relies on the following standard assumptions.

- **KEM IND-CCA Security.** The key encapsulation mechanism used in the MFK exchange satisfies the IND-CCA security notion. For any adversary $\mathcal{A}$, the advantage in distinguishing the encapsulated shared secret from a random value is negligible. We denote this advantage by $\mathsf{Adv}_{\mathsf{KEM}}(\mathcal{A})$.

Primitive	Security Property	Advantage
KEM	IND-CCA confidentiality	$\mathsf{Adv}_{\mathsf{KEM}}(\mathcal{A})$
Signature scheme	EUF-CMA unforgeability	$\mathsf{Adv}_{\mathsf{SIG}}(\mathcal{A})$
SHAKE / Keccak	PRF or RO indistinguishability	$\mathsf{Adv}_{\mathsf{PRF}}^{\mathrm{SHAKE}}(\mathcal{A})$
KMAC / AEAD layer	Ciphertext integrity	$\mathsf{Adv}_{\mathrm{AEAD}}(\mathcal{A})$
Keccak based KDF	Extraction from entropic inputs	$\frac{q_h}{2^{256}}$
AEAD header binding	Replay resistance	negligible

Table 2: Security assumptions used in the MPDC-I reduction proofs and their associated advantage notation.

- **Signature EUF-CMA Security.** The signature scheme used for certificate authentication and MFK transcript signing is existentially unforgeable under chosen message attack. For any adversary $\mathcal{A}$ that outputs a valid signature on a new message, the probability of success is negligible and is denoted $\mathsf{Adv}_{\mathsf{SIG}}(\mathcal{A})$.
- **SHAKE Based pseudo-randomness.** The SHAKE based derivation functions and the Keccak sponge used in fragment masking and KDF operations are treated as pseudo-random functions or as a random oracle. We denote the advantage of distinguishing SHAKE output from random by $\mathsf{Adv}_{\mathsf{PRF}}^{\mathrm{SHAKE}}(\mathcal{A})$.
- **AEAD Security.** The authenticated encryption layer used in the tunnel provides confidentiality and integrity under unique associated data. Any adversary that forges a valid ciphertext and tag pair has negligible advantage, which we denote $\mathsf{Adv}_{\mathrm{AEAD}}(\mathcal{A})$.

All reductions will be expressed in terms of these advantages.

6.2 Confidentiality of Master Fragment Keys

Theorem 6.1 (Security of MFK Exchange). *Any adversary that obtains a non negligible advantage against the confidentiality of a Master Fragment Key in MPDC-I yields an adversary of comparable advantage against the IND-CCA security of the KEM or the EUF-CMA security of the signature scheme.*

Proof. Consider an adversary $\mathcal{A}$ that distinguishes an MFK value from a random value with non negligible advantage. We construct a reduction $\mathcal{B}$ that uses $\mathcal{A}$ to break the KEM or the signature scheme.

Embedding the KEM Challenge. The reduction $\mathcal{B}$ receives a KEM challenge and embeds it into the MFK exchange for one of the device pairs. All other exchanges are simulated honestly. If $\mathcal{A}$ distinguishes the real MFK from random then it distinguishes the challenge shared secret with the same probability. Thus $\mathcal{B}$ wins the IND-CCA game whenever $\mathcal{A}$ wins.

Authentication Control Through Signatures. To prevent $\mathcal{A}$ from injecting malicious public keys or modifying transcript data, the reduction simulates all signatures using a signing oracle. If $\mathcal{A}$ forges a valid signature on a new message, $\mathcal{B}$ outputs the forgery and wins the EUF-CMA game.

Conclusion. Any non negligible advantage against MFK confidentiality leads to a non negligible advantage against either the KEM or the signature scheme. Since both primitives are assumed secure, MFK confidentiality is preserved. $\square$

6.3 Indistinguishability of Tunnel Keys

Theorem 6.2 (Tunnel Key Indistinguishability). *For any efficient adversary $\mathcal{A}$ we have:*

$$\mathsf{Adv}^{\mathsf{AKE}}_{\mathsf{MPDC}}(\mathcal{A}) \leq \mathsf{Adv}_{\mathsf{KEM}}(\mathcal{B}_{\mathsf{KEM}}) + \mathsf{Adv}_{\mathsf{SIG}}(\mathcal{B}_{\mathsf{SIG}}) + \mathsf{Adv}^{SHAKE}_{\mathsf{PRF}}(\mathcal{B}_{\mathsf{PRF}}) + \frac{q_h}{2^{256}},$$

where q_h is the total number of hash or random oracle queries and the $\mathcal{B}$ are efficient reductions.

Proof. The proof proceeds using a standard hybrid argument.

Hybrid H1: Replace MFK Values with Random. Replace all MFK values with random strings. Any adversary that distinguishes this hybrid from the real world yields $\mathcal{B}_{\mathsf{KEM}}$ breaking the KEM.

Hybrid H2: Replace Fragment Masking Keystreams. Replace SHAKE based keystreams with random strings. Detecting this replacement yields $\mathcal{B}_{\mathsf{PRF}}$ that breaks SHAKE pseudo-randomness.

Hybrid H3: Replace KDF with a Random Oracle. Replace the Keccak based KDF with a random oracle. By the entropy lemma for one honest Agent, the adversary can guess the full KDF input only with probability $q_h/2^{256}$.

Hybrid H4: Replace the Tunnel Key. Replace the KDF output with a uniform random value. Any distinguisher yields a random oracle adversary $\mathcal{B}_{\mathsf{PRF}}$.

Conclusion. Each hybrid change is indistinguishable under the stated assumptions, proving the claimed bound. □

6.4 Integrity and Replay Resistance

Theorem 6.3 (Ciphertext Integrity and Replay)**.** *Any adversary that causes an honest party to accept a new ciphertext or to accept a replayed ciphertext with non negligible probability yields an adversary of comparable advantage against the AEAD or KMAC based authentication layers.*

Proof. We consider both properties.

Ciphertext Integrity. An adversary succeeds only if it forges a valid ciphertext and tag pair not output by any encryption oracle. Sequence numbers and timestamps are included in the associated data, so any header or ciphertext change invalidates the tag. A successful adversary therefore yields a forgery against the AEAD layer.

Replay Resistance. A replay succeeds only if the adversary causes an honest party to accept the same message twice. Because sequence numbers are monotonic and timestamps must lie within a valid window, acceptance of a replay requires forging a new valid tag under altered associated data, giving a MAC forgery against the KMAC layer.

Conclusion. Both properties follow from the integrity of the AEAD layer and the unforgeability of KMAC. Under the assumptions above, both advantages are negligible. □

7 Trust Anchor and Topology Analysis

This section analyzes the role of the RDS root certificate, the security implications of DLA compromise, and the consistency of topology information accepted by honest parties. Trust anchors are central to MPDC-I because certificate validation, topology distribution, and revocation all depend on authenticated messages rooted in the RDS key.

Entity	Signing Key	Issuer	Topology	MFK
RDS	Yes	Self	No	No
DLA	Yes	RDS	Yes	No
MAS	Yes	RDS	No	Yes
Client	Yes	RDS	No	Yes
Agent	Yes	RDS	No	Yes

Table 3: Trust anchors and device roles in MPDC-I. RDS acts as the sole certificate root, DLA distributes topology and revocation information, and MAS, Client, and Agents participate in authenticated MFK establishment and fragment collection.

7.1 RDS Security Properties

The RDS root certificate authority is the only entity permitted to sign device certificates and the DLA certificate. Its public key is embedded in every device and defines the trust boundary of the entire domain. A certificate is accepted by a device only if its signature chain verifies back to the RDS root.

The uniqueness of the RDS key ensures that no adversary can introduce arbitrary devices or management authorities into the domain. Since no entity other than RDS can issue a valid certificate, the adversary cannot impersonate a Client, MAS, or Agent without forging a signature under the RDS key. This reduces impersonation and device insertion attacks to the unforgeability of the signature scheme used in MPDC.

Topology messages, revocation lists, and configuration updates are signed by the DLA but are accepted only when the DLA certificate verifies under the RDS root. Thus the authenticity of topology and revocation information ultimately depends on the RDS key. As long as the RDS key remains uncompromised, no adversary can forge a certificate or introduce a new DLA instance.

7.2 DLA Compromise and Its Effects

Lemma 7.1 (DLA Compromise Does Not Break Authentication). *If the DLA is compromised, the adversary can prevent progress in the system and can misroute or suppress topology updates, but cannot forge valid device certificates or break tunnel authentication without either the RDS signing key or the long term signing keys of Client, MAS, or Agents.*

Proof. Assume the adversary compromises the DLA and learns its signing key. The adversary can now produce valid DLA signed topology and revocation messages. We show that this does not grant the adversary any additional power to break authentication or confidentiality of the tunnel.

Device Certificates. All device certificates are signed under the RDS key. The DLA has no authority to issue device certificates. Any attempt to introduce a forged device certificate therefore requires forging a signature under the RDS key. By the EUF-CMA security of the signature scheme, this probability is negligible.

MFK Exchanges. The DLA is not involved in MFK exchange. These exchanges are authenticated only through the device certificates verified under the RDS root. Since the DLA key does not participate in these operations, compromise of the DLA has no effect on MFK confidentiality or authenticity.

Tunnel Keys. Tunnel keys depend only on authenticated fragment transmissions between Client, MAS, and Agents. These transmissions are protected by device signatures on request messages and by KMAC under MFK derived keys on responses. The adversary cannot forge valid fragment messages without compromising the corresponding device signing keys or breaking KMAC.

Conclusion. A compromised DLA can degrade availability by manipulating topology, but cannot influence authentication or break tunnel secrecy. These properties reduce to the unforgeability of the signature scheme and the fact that all device certificates chain to the RDS root. $\square$

7.3 Topology Consistency Lemma

Lemma 7.2 (Consistency of Accepted Topology). *If two honest parties accept a topology update, then their view of the active Agent set is consistent enough to prevent partition attacks or downgrade attacks under the system adversary model.*

Proof. Let T be a topology update accepted by two honest devices D_1 and D_2. To be accepted, the topology update must satisfy the following conditions for each device.

- The update is signed by the DLA certificate.
- The DLA certificate chains to the RDS root.
- The sequence number of the update exceeds the stored sequence number.

- The timestamp lies within the permitted window.

The first two conditions ensure that both devices validate the same DLA key. Since both devices validate the DLA certificate under the RDS root, the adversary cannot present different DLA credentials to different parties.

Sequence monotonicity ensures that neither device accepts an older topology. If D_1 and D_2 accept the same update T, then T must be fresher than their previous state and thus both transition to the same topology version.

The timestamp check ensures freshness with respect to local clocks. This prevents an adversary from supplying stale updates to one device and fresh updates to another, since stale updates fail the timestamp check.

These checks guarantee that two honest devices accepting the same topology update record an equivalent view of the active Agent set. Because the active Agent set determines which fragments must be reconstructed for each session, consistent topology ensures that the Client and MAS include the same Agents in fragment reconstruction, preventing partition or downgrade attacks.

Therefore topology acceptance remains consistent across honest devices under the stated assumptions. □

8 Post Compromise Security

This section formalizes the notion of post compromise security for MPDC-I and proves that the protocol recovers full confidentiality after a compromise as long as the next fragment collection round includes at least one honest Agent. This property is a direct consequence of the distributed entropy design and the entropy contribution of honest Agents established earlier.

8.1 PCS Model for MPDC

Post compromise security captures the ability of a protocol to recover secrecy after an adversary has learned all secrets known to a party at some point in time. We consider compromise of the Client or the MAS, since these parties hold the tunnel keys. A compromise event reveals all long term keys and all session keys derived up to the time of compromise. The adversary may also observe all fragments transmitted during the compromised round and may inspect all associated transcripts.

Recovery occurs only during the next fragment collection round. Let t be the time of compromise and let $t+1$ be the time of the next session establishment. Let $\mathcal{A}_{t+1}$ be the set of Agents participating in the fragment collection at time

$t+1$. The fragment of an honest Agent in this set is sampled independently at time $t+1$ and is unknown to the adversary.
A protocol provides PCS if the adversary cannot distinguish the fresh session key generated at time $t+1$ from a random key, even after learning all secrets known at time t.

Formally, in the PCS game the adversary:

- corrupts the Client or the MAS at time t,
- learns all long term keys and all session keys generated up to time t,
- observes the fragment collection round at time $t+1$, and
- receives either the real session key SK_{t+1} or a random value of the same length.

The adversary must guess which value it received. The advantage in this game is denoted $\mathsf{Adv}^{\mathsf{PCS}}_{\mathsf{MPDC}}(\mathcal{A})$.

8.2 Healing Lemma

Lemma 8.1 (Post Compromise Healing). *Let the adversary compromise the Client or MAS at time t and learn all secrets known at that time. Suppose the fragment collection round at time t* $+ 1$ *includes at least one honest Agent whose fragment is uniformly distributed and independent of the adversary view at time t. Then the session key derived at time t* $+ 1$ *is independent of all adversarially known material at time t and is computationally indistinguishable from a uniform key. In particular,* $\mathsf{Adv}^{\mathsf{PCS}}_{\mathsf{MPDC}}(\mathcal{A})$ *is negligible.*

Proof. At time t the adversary learns all long term secrets and all previously derived session keys. It also obtains full transcripts of all fragment exchanges up to time t. However, the adversary does not gain the ability to forge signatures or MACs under uncompromised devices and does not learn future randomness generated after time t.
At time $t+1$ a new fragment collection round occurs. Let F^* denote the fragment generated by an honest Agent in this round. By definition of compromise, F^* is sampled after time t and is therefore independent of the adversary view at time t. The adversary observes only the masked version of F^* together with a KMAC tag. Since the ephemeral fragment key and the associated MFK for the honest Agent remain unknown to the adversary, the adversary does not learn F^* and cannot recover it from the masked fragment.

By the entropy lemma for one honest Agent, the aggregate value

$$h_{t+1} = \mathsf{SHAKE}\left(F_{\mathrm{MAS}} \parallel F_1 \parallel \cdots \parallel F^*\right)$$

has min entropy at least $256 - \varepsilon$ conditioned on the adversary view, where ε is negligible. This remains true even if all other fragments are adversarially chosen.

The tunnel keys at time $t + 1$ are derived from h_{t+1} as

$$K_{\text{tx}} = \mathsf{SHAKE}(h_{t+1} \,\|\, 01), \qquad K_{\text{rx}} = \mathsf{SHAKE}(h_{t+1} \,\|\, 02).$$

Under the pseudo-randomness of SHAKE, if h_{t+1} has high min entropy then the key pair $(K_{\text{tx}}, K_{\text{rx}})$ is indistinguishable from uniform. Since h_{t+1} is independent of all adversarially known material at time t, the derived keys are also independent of all prior secrets.
Therefore the adversary cannot distinguish the real session key at time $t + 1$ from a random one except with negligible advantage, proving the PCS property.
□

9 Implementation Conformance and System Considerations

This section states the implementation level assumptions that are required for the formal proofs to hold. These assumptions reflect the behavior of the reference implementation and are consistent with the threat model established in Section 3. They also ensure that the reductions used in the security analysis correspond to the behavior of the deployed protocol.

9.1 Constant Time Behavior and Side Channels

The security model assumes that all cryptographic operations that depend on secret data execute in constant time. In particular, the following operations must not leak timing information.

- decapsulation and shared secret derivation in the KEM used for MFK exchange,
- signature verification during certificate validation and MFK transcript authentication,
- RCS or AES-GCM encryption and MAC verification,
- SHAKE based key derivation and fragment masking.

Any timing deviation that depends on secret values could enable side channel attacks that lie outside the formal model and would invalidate proofs that rely on IND-CCA or EUF-CMA security. The reference implementation satisfies these constraints by using constant time routines for KEM decapsulation, signature verification, and RCS or AES-GCM operations. We assume that all compliant implementations retain this property.
The model further assumes that no side channel is available through memory access patterns, error messages, or conditional branches. All authenticated decryption failures in the tunnel AEAD layer must terminate the tunnel without revealing additional information about the cause of the failure.

9.2 Random Number Generation Requirements

The security of MPDC-I depends on the availability of high quality randomness at each device. The following assumptions are required.

- Each device uses a cryptographically secure random number generator that provides full entropy for all calls.
- Each Agent samples its fragment F_i using the system random generator or SHAKE based expansion so that F_i is uniformly distributed on $\{0,1\}^{256}$.
- The Client and MAS generate any ephemeral KEM keys required for MFK establishment using the same class of secure RNG.

The entropy lemma requires that the fragment from at least one honest Agent be sampled uniformly and independently of the adversary. If the RNG at an honest Agent fails or produces biased output, the guarantees of Section 4 may not hold. The model therefore assumes that RNGs are properly seeded, that reseeding occurs as needed, and that device operators ensure ongoing RNG health.

9.3 Sequence Numbers, Timestamps, and Failure Handling

The replay and integrity proofs in Section 5 rely on the concrete structure of the tunnel header. This header contains a function code, a ciphertext length, a sequence number, and a timestamp. These fields are incorporated into the associated data of the AEAD invocation.
The implementation uses the following rules, which are required for the security argument.

- **Sequence numbers.** Each direction maintains a 64 bit monotonic sequence number that starts at zero when a tunnel is established and increments by one for each outbound message. The receiver accepts a ciphertext only if the sequence number matches the expected value.
- **Timestamps.** Each message contains a timestamp measured in seconds. A message is accepted only if its timestamp differs from the local clock by at most a fixed window $T_{\max}$, typically sixty seconds. Messages outside this window are rejected before any decryption operation.
- **Associated data.** Both the sequence number and the timestamp are included in the associated data of the AEAD invocation, so any modification of these values causes tag verification to fail.
- **Failure handling.** If any header check fails or if AEAD verification fails, the receiver terminates the tunnel and clears all state. No partial plaintext is released and no error code dependent on secret information is returned.

These rules ensure that an adversary cannot cause an honest device to accept a replayed message or a modified message without producing a valid MAC. The formal replay game in Section 5 models these exact conditions. A replay attack requires either duplication of the same sequence number or a forgery of the AEAD tag, both of which have negligible probability under the assumed AEAD security.
Together these considerations ensure that the behavior of the implementation matches the assumptions of the formal model and that the reductions in the security analysis reflect real deployment conditions.

10 Conclusion

This paper presented a complete provable security analysis of MPDC-I, a multiparty key establishment and secure tunnel protocol that derives its secrecy from distributed entropy across a set of independent Agents. The analysis followed the structure of the normative specification and provided a formal execution model, a security experiment specific to MPDC, an entropy lemma for one honest Agent, and reduction based proofs that cover confidentiality, integrity, replay resistance, trust anchor properties, and post compromise recovery.

10.1 Summary of Results

We formalized the roles of Client, MAS, Agents, DLA, and RDS and described the certificate hierarchy, topology distribution mechanism, and authenticated KEM based establishment of Master Fragment Keys. We introduced an engineering description of MPDC-I that accurately reflects the specification and serves as the basis for the formal model.

We defined a dedicated MPDC Indistinguishability experiment that includes corruption oracles for all trust relevant entities. This model captures compromise scenarios that are not addressed in standard two party authenticated key exchange definitions. Within this framework we proved an entropy lemma showing that a single honest Agent fragment ensures that the aggregated hash used for key derivation retains full min entropy, even if all other fragments are adversarially controlled.

Using this lemma we presented reduction based proofs for the secrecy of Master Fragment Keys, the indistinguishability of tunnel keys, and the integrity and replay resistance of the encrypted tunnel. The reductions give concrete advantage bounds in terms of the security of the KEM and signature primitives used for MFK establishment and certificate authentication, along with SHAKE based extraction. We also analyzed the effects of DLA compromise and proved that such compromise cannot break authentication or confidentiality without violating the unforgeability of the signature scheme. Finally, we formalized post compromise security for MPDC and proved that the protocol heals in the next fragment collection round whenever at least one Agent remains honest.

10.2 Limitations

The security results in this paper rely on several idealized assumptions. We assume that SHAKE behaves as a pseudo-random function or a random oracle and that Keccak based extraction behaves ideally. We also assume that the RNGs used by devices are secure and that the implementation executes all secret dependent operations in constant time. Although the reference implementation satisfies these properties, side channel leakage or insufficient entropy in a deployment could undermine the security guarantees.

Our model assumes that at least one Agent remains uncompromised during each fragment collection round. If all Agents are compromised or if the topology is manipulated to exclude all honest Agents, the entropy guarantees do not hold and the protocol provides no confidentiality in this degenerate scenario. Furthermore, availability is not treated in the formal model. A fully compromised DLA can prevent topology updates or suppress fragment collection rounds, which may impact liveness even though confidentiality and authentication remain intact.

10.3 Directions for Future Work

There are several directions in which the formal analysis of MPDC can be extended. One direction is to develop a tighter reduction for the SHAKE based extraction process using recent results on indifferentiability for sponge based constructions. Another direction is to incorporate availability properties and fault tolerant behavior into the model to analyze denial of service and partial failure scenarios. It may also be valuable to explore variants of MPDC that use larger Agent sets or dynamic Agent membership and to study their effect on key derivation and trust anchor consistency. Finally, integrating formal verification tools or symbolic models with the computational analysis in this paper may yield complementary insights and support automated validation of future protocol extensions.

Overall, this work establishes a rigorous foundation for understanding the security of MPDC-I and provides a framework for analyzing distributed entropy based protocols in similar trust managed environments.

References

1. National Institute of Standards and Technology (NIST). *FIPS 202: SHA–3 Standard, Permutation-Based Hash and Extendable-Output Functions.* U.S. Department of Commerce, 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf
2. National Institute of Standards and Technology (NIST). *FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM).* U.S. Department of Commerce, 2024. https://csrc.nist.gov/pubs/fips/203/final
3. National Institute of Standards and Technology (NIST). *FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA).* U.S. Department of Commerce, 2024. https://csrc.nist.gov/pubs/fips/204/final
4. Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. *Sponge Functions.* ECRYPT Hash Function Workshop, 2007. https://keccak.team/files/SpongeFunctions.pdf
5. Bertoni, G., Daemen, J., Peeters, M., and Van Assche, G. *The Keccak Reference.* Submission to the NIST SHA–3 Competition, 2011.
6. Keccak Team. Reference software and documentation. https://keccak.team
7. QRCS Corporation. Project resources and supplementary materials. https://www.qrcscorp.ca
8. QRCS Corporation. *multi-party Domain Cryptosystem Technical Specification, MPDC 1.0.* Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://www.qrcscorp.ca/documents/mpdc_specification.pdf
9. Underhill, J. *multi-party Distributed Cryptosystem – Technical Specification, MPDC 1.0.* Quantum Resistant Cryptographic Solutions Corporation, 2025. Available at: https://www.qrcscorp.ca/documents/mpdc_specification.pdf
10. QRCS Corporation. MPDC Reference Implementation (Source Code Repository). https://github.com/QRCS-CORP/MPDC
11. Underhill, J. *RCS Specification.* Quantum Resistant Cryptographic Solutions Corporation, 2021. https://www.qrcscorp.ca/documents/rcs_specification.pdf