Individual Submission
Internet-Draft
Intended status: Standards Track
Expires: 19 August 2026

S. Verma

15 February 2026

A Capability-Oriented Intent Routing Protocol
draft-verma-cirp-00

Abstract

This document specifies a transport-layer protocol for routing capability-oriented intent between cryptographically identified endpoints across heterogeneous trust domains. The protocol defines capability identifiers with mandatory versioning, scoped discovery across five visibility levels, ticket-based session authorization, negotiated cryptographic suites including hybrid post-quantum key establishment, encrypted peer-to-peer session establishment, and mutually attested invocation receipts with hash-chained integrity. Payload semantics are opaque to the transport layer.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 19 August 2026.

Copyright Notice

Table of Contents

1. Introduction

Existing transport protocols provide general-purpose data delivery between network endpoints. They do not address the requirements of systems that need to discover capabilities by function, route structured invocations to providers of those capabilities, and produce cryptographically verifiable records of fulfillment. These requirements arise in domains including autonomous systems, industrial control, medical device coordination, software service composition, and other environments where heterogeneous endpoints must locate and invoke capabilities across trust boundaries.

The Capability-Oriented Intent Routing Protocol (CIRP) is a transport-layer protocol that addresses these requirements. It provides: structured capability identifiers with mandatory versioning; scoped discovery that respects organizational and cryptographic trust boundaries; ticket-based session authorization that removes the authorizing registry from the data path; encrypted peer-to-peer sessions with cryptographic agility including hybrid post-quantum key exchange; typed invocation envelopes with opaque payloads; and mutually attested fulfillment receipts.

CIRP operates at the transport layer. It does not interpret payload semantics, evaluate business logic, rank providers, manage economic relationships, or implement domain-specific processing. These concerns belong to application layers built above the transport. The protocol is designed to carry executable intent for any domain: the payload may contain JSON, binary protocol encodings, robotics command sequences, medical device instructions, industrial control messages, or any other octet sequence. The concept of an "Internet of Intent" describes a network infrastructure where such invocations can be routed across heterogeneous trust domains using a common transport protocol.

This document specifies the protocol messages, encoding formats, cryptographic operations, and security properties of CIRP. It defines an initial transport binding for UDP and establishes IANA registries for cryptographic suites, capability URI schemes, and protocol parameters.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Non-Goals

The following topics are explicitly outside the scope of this specification:

* Orchestration, coordination, or workflow semantics between endpoints.
* Economic models, billing, pricing, or monetization of capabilities.
* Reputation, ranking, or quality-of-service scoring of providers.
* Capability namespace governance beyond the reserved prefix defined in Section 5.4.
* Registry federation synchronization protocol. (Federation is acknowledged but defined in a companion specification.)
* Application payload interpretation, validation, or transformation.
* NAT traversal mechanisms beyond registry-mediated locator distribution.

2. Terminology

Intent A cryptographically signed, versioned invocation of a capability that is routable over the network and produces a verifiable fulfillment response.

Capability A named, versioned unit of functionality addressable via a structured URI (Section 5).

Endpoint Identifier (EID) A node’s cryptographic identity, equal to its public verification key.

Registry Infrastructure that provides capability discovery and session authorization. Registries do not participate in data plane communication.

Consumer The party that initiates an invocation.

Provider The party that fulfills an invocation.

ConnectTicket A signed, time-bounded authorization artifact issued by a Registry that permits a Consumer to establish a direct session with a Provider.

Trust Domain A set of identities sharing a common trust anchor, used to restrict capability visibility.

Suite A named combination of cryptographic algorithms used for key exchange, authentication, encryption, and key derivation within a session.

Fulfillment Record A dual-signed receipt attesting that a specific invocation was processed and fulfilled, constructed cooperatively by Provider and Consumer (Section 10.1).

Scope A visibility descriptor attached to a capability advertisement that restricts which Consumers may discover it. Five scope levels are defined (Section 6.3).

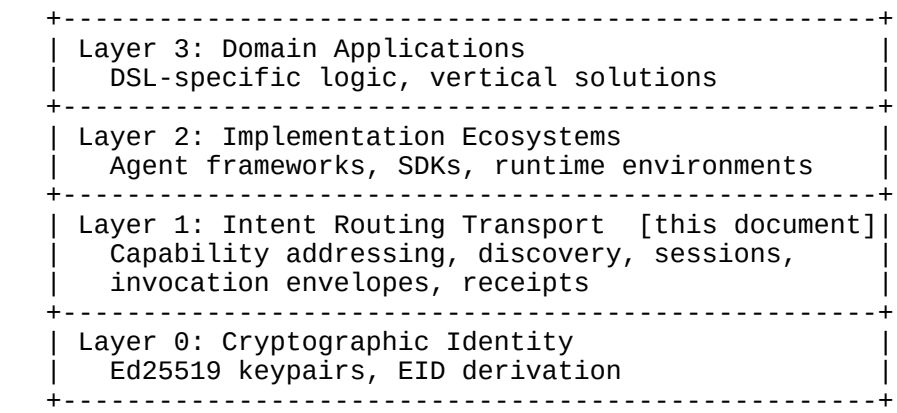
Policy Receipt A diagnostic record included in discovery responses when the Registry applies policy that modifies the result set.

3. Architecture Overview

CIRP defines a layered architecture for routing capability-oriented intent between cryptographically identified endpoints. The architecture separates concerns into distinct layers, each with well-defined responsibilities and trust boundaries.

3.1. Layer Model

The protocol operates at Layer 1 in the following conceptual stack:



This document specifies Layer 1. Layers above the transport are unconstrained by this specification: any agent framework, domain-specific language, or application architecture may operate over CIRP provided it uses the defined capability addressing, session establishment, and invocation envelope formats.

Layer 0 is the identity foundation. An Endpoint Identifier (EID) is the Ed25519 public key of a node. Because EID equals the public key, identity verification requires no certificate authority, no public key infrastructure, and no trusted third party. A node proves its identity by signing messages with the corresponding private key.

3.2. Protocol Roles

Three roles participate in the protocol:

Registry A Registry provides two services: capability discovery (mapping capability identifiers to Provider endpoints) and session authorization (issuing ConnectTickets that permit direct peer-to-peer sessions). A Registry does not participate in data plane communication and never observes invocation content or session keys. Multiple Registries MAY operate concurrently, and endpoints MAY interact with different Registries for different capabilities.

Consumer A Consumer discovers capabilities through a Registry, obtains authorization via a ConnectTicket, and initiates a direct session with a Provider to send invocations and receive fulfillments.

Provider A Provider advertises capabilities to one or more Registries via presence announcements, accepts authorized sessions from Consumers, processes invocations, and returns fulfillment responses.

A single endpoint MAY act as both Consumer and Provider simultaneously for different capabilities. The protocol does not impose a fixed client-server relationship; any endpoint with a cryptographic identity may assume either role.

3.3. Trust Assumptions

The protocol operates under the following trust model:

Endpoints are self-sovereign Each endpoint generates its own Ed25519 keypair and derives its EID from the public key. No registration authority or certificate issuer is required. Identity is established through cryptographic proof, not through delegation.

Registries are semi-trusted Registries are trusted to perform discovery and authorization honestly, but they are not trusted with session content. The protocol is designed so that a compromised or malicious Registry cannot forge endpoint identities, decrypt data plane traffic, or tamper with invocation records. A detailed analysis of what a Registry can and cannot do is provided in Section 11.

The network is untrusted The protocol assumes an active network attacker who can observe, inject, modify, and replay messages. All data plane communication is encrypted and authenticated. Control plane messages are signed. Tickets are bound to specific identities and time-limited.

4. Transport Model

CIRP separates protocol operations into two planes with distinct security properties and operational characteristics.

4.1. Control Plane

The control plane handles registry-mediated operations: presence registration, capability advertisement, discovery queries, admission control, and ConnectTicket issuance. Control plane messages are exchanged between endpoints and registries.

Control plane messages are small, fixed-structure, and designed to fit within a single transport datagram for typical deployments. This design prioritizes low-latency registry interactions and avoids fragmentation on common network paths.

The control plane does not carry invocation content. It handles only the metadata necessary for discovery and session authorization.

4.2. Data Plane

The data plane handles peer-to-peer communication between consumer and provider after session establishment. All data plane traffic is encrypted using session keys derived during key exchange (Section 7.4). The registry is not involved in data plane operations: it does not relay data, does not possess session keys, and cannot observe invocation content.

The data plane carries invocation envelopes, response envelopes, and fulfillment records as defined in Section 9 and Section 10. Payload content within these envelopes is opaque to the transport.

4.3. Transport Binding

This specification defines the protocol abstractly in terms of messages exchanged between roles. A transport binding maps these abstract messages onto a concrete network transport.

The initial transport binding is UDP. Control plane messages are exchanged as UDP datagrams between endpoints and registries. Data plane encrypted frames are exchanged as UDP datagrams between peers. Implementations are responsible for handling MTU constraints and chunking large payloads into multiple datagrams as needed.

The protocol does not rely on TCP semantics. It does not assume ordered delivery, connection state, or stream-oriented framing at the transport layer. Each protocol message is self-contained within its datagram.

Future transport bindings (for example, QUIC) may be defined in companion specifications. The protocol's message semantics are transport-independent; only the framing and delivery characteristics change with the binding.

5. Capability Identifiers and Versioning

Capabilities in CIRP are addressed using structured URIs that combine a hierarchical namespace with mandatory version information. This addressing scheme enables precise discovery, prevents semantic drift across protocol versions, and supports independent evolution of capabilities within distinct organizational namespaces.

5.1. URI Syntax

A capability URI uses the "cap" scheme and has the following structure:

```
cap-uri   = "cap:" path "/" version
path      = segment 1*("." segment)
segment   = ALPHA *(ALPHA / DIGIT / "-")
version   = "v" major "." minor
major     = 1*DIGIT
minor     = 1*DIGIT
```

The path component consists of dot-delimited segments forming a hierarchical namespace. The grammar requires at least two segments (one dot), establishing a minimum structure of namespace.action. Deeper hierarchies are permitted (e.g., "acme.robotics.arm.wave").

Segments MUST begin with an ASCII letter and MAY contain ASCII letters, digits, and hyphens. Segments are case-sensitive.

Examples of valid capability URIs:

```
cap:echo.ping/v1.0
cap:robot.wave/v1.0
cap:acme.robotics.arm.wave/v2.1
cap:org.medical.imaging.analyze/v1.0
```

Examples of invalid capability URIs:

```
cap:echo/v1.0          (single segment, no dot)
cap:robot.wave         (missing version)
cap:robot.wave/1.0     (version missing "v" prefix)
cap:123.test/v1.0      (segment begins with digit)
```

5.2. Versioning

Every capability URI MUST include a version component in the format "v" followed by major.minor integers. Both the major and minor version numbers are REQUIRED.

Version matching in this specification uses exact semantics: a Provider advertising cap:x.y/v1.2 MUST NOT fulfill requests for cap:x.y/v1.3 unless the Provider explicitly advertises that version as well. Consumers MUST include the complete version in all discovery requests and invocation envelopes.

Compatible-range version matching (where a request for v1.2 could be fulfilled by a Provider advertising v1.3) is not defined in this specification. Future revisions MAY define version compatibility semantics.

5.3. Capability Hashing

For wire efficiency, capabilities are referenced in protocol messages by their hash rather than their full URI string. The capability hash is computed as follows:

1. Take the complete capability URI as a UTF-8 byte string (e.g., "cap:robot.wave/v1.0").

2. Compute the SHA-256 digest of this byte string.

3. The resulting 32-byte value is the capability hash.

Capability hashes are used in Authorization Requests (Section 7.2), ConnectTickets (Section 7.2.1), and presence advertisements. The full URI string is carried only in invocation envelopes (Section 9) where human readability and version information are required.

5.4. Namespace Considerations

The prefix "cap:proto." is RESERVED for capabilities defined by this protocol and its companion specifications. Implementations MUST NOT advertise capabilities under the "proto" namespace unless those capabilities are defined in a CIRP specification document.

Namespace allocation and governance beyond the reserved prefix is a deployment concern. A registry for well-known capability namespaces may be established in a future revision of this specification. Deployments SHOULD use organizational domain names in reverse notation (e.g., "com.example.service.action") to minimize namespace collisions.

6. Discovery

Discovery is the process by which a Consumer locates Providers of a given capability. The protocol defines a query-response mechanism mediated by Registries and a scoping model that controls capability visibility across organizational and trust boundaries.

At the protocol layer, discovery is unfiltered within the applicable scope: Registries do not rank, suppress, or prioritize Providers beyond the scope restrictions declared in capability advertisements. Visibility is governed by scope, not by policy ranking or commercial preference.

6.1. Discovery Queries

A Consumer discovers Providers by sending a Discovery Query to a Registry. The query contains:

capability_hash (32 bytes) The SHA-256 hash of the desired capability URI, computed as specified in Section 5.3.

max_results (2 bytes, unsigned integer, big-endian) The maximum number of Providers to return.

locality_hint (8 bytes, optional) A geographic locality indicator that the Registry MAY use to prefer geographically proximate Providers. The locality hint is advisory; the Registry is not required to honor it.

The Consumer does not need to be authenticated to submit a Discovery Query, but the Registry MAY require the Consumer to be in an admitted state depending on deployment policy.

6.2. Discovery Responses

The Registry responds to a Discovery Query with a Discovery Response containing zero or more Provider entries. Each entry includes:

provider_eid (32 bytes) The Provider's Endpoint Identifier.

provider_locator (7 or 19 bytes) The Provider's network locator, consisting of an address type indicator (1 byte: 0x04 for IPv4, 0x06 for IPv6), a port number (2 bytes, big-endian), and the IP address (4 or 16 bytes). The locator reflects the Provider's observed network address as determined by the Registry, not a self-reported value.

The Registry uses the Provider's observed network address (the source address of the Provider's most recent presence message) as the authoritative locator. Self-reported addresses in presence messages are not used for routing, preventing stale or spoofed locators from directing traffic to incorrect endpoints.

Providers whose most recent presence beacon is older than a deployment-configured freshness threshold are excluded from query results. The freshness check uses the original beacon timestamp as recorded by the originating Registry, not timestamps assigned during inter-Registry synchronization. This prevents synchronized records from appearing artificially fresh.

When a Registry applies policy that modifies the result set (e.g., scope restrictions or tier-based limits), the response SHOULD include a Policy Receipt indicating the nature and reason for the modification. This supports the principle that policy enforcement is permitted but silent policy enforcement is not: consumers and diagnostic tools can observe how policies affect discovery results.

6.3. Capability Scoping

Capability advertisements carry a scope descriptor that controls their visibility during discovery. Scope enforcement is performed by the Registry during query processing, not by peer-side filtering logic. This ensures that scope is enforced consistently regardless of Consumer implementation.

Five scope levels are defined:

6.3.1. Scope Levels

Level	Value	Qualifier	Semantics
Local	0x00	None	Visible only to the advertising endpoint itself. Used for internal bookkeeping or credential-agent patterns where a capability serves only its owner.
Identity	0x01	target_eid (32 bytes)	Visible only to the endpoint identified by target_eid. Used for pre-arranged point-to-point capabilities.
Trust Domain	0x02	anchor_hash (32 bytes)	Visible to endpoints whose identity chain includes the trust anchor identified by anchor_hash.
Organization	0x03	anchor_hash (32 bytes)	Visible to endpoints within the organization identified by anchor_hash. Semantically equivalent to Trust Domain but conventionally used for organizational boundaries.
Global	0x04	None	Visible to all endpoints without restriction.

Table 1

6.3.2. Scope Encoding

The scope descriptor is carried in capability advertisements as a compact binary encoding:

Scope Descriptor

Offset	Size	Field	
0	1	scope_level	
1	0/32	qualifier (present for levels 0x01-0x03)	

Total size:

- Global (0x04): 1 byte
- Local (0x00): 1 byte
- Identity (0x01): 33 bytes (1 + 32-byte target EID)
- Trust Domain (0x02): 33 bytes (1 + 32-byte anchor hash)
- Organization (0x03): 33 bytes (1 + 32-byte anchor hash)

The anchor_hash for Trust Domain and Organization scopes is the SHA-256 hash of the trust anchor's Ed25519 public key. This allows scope enforcement without transmitting the full public key in every advertisement.

6.3.3. Scope Enforcement

When processing a Discovery Query, the Registry evaluates the scope of each candidate Provider's capability advertisement against the querying Consumer's identity:

Global No restriction. The capability is included in results for any Consumer.

Organization / Trust Domain The Registry checks whether the Consumer's identity chain includes the trust anchor identified by anchor_hash. If not, the capability is excluded from results.

Identity The Registry checks whether the Consumer's EID matches the target_eid in the scope qualifier. If not, the capability is excluded from results.

Local The capability is never included in Discovery Responses. Local capabilities are visible only through the advertising endpoint's own internal interfaces.

The mechanism by which a Registry determines membership in a trust domain is deployment-specific. Possible approaches include signed membership assertions from the trust anchor, pre-configured identity-to-domain mappings, or delegation chains. This specification defines the scope encoding and enforcement semantics but does not mandate a specific trust domain membership protocol.

6.4. Federated Discovery

In deployments with multiple Registries, Providers may register with different Registries. To provide a consistent view of available capabilities, Registries MAY synchronize presence information through a federation protocol.

When a Registry includes federated records in Discovery Responses, the freshness of those records MUST be evaluated using the original beacon timestamp as recorded by the Provider's home Registry, not the timestamp at which the record was received via federation. This prevents synchronized records from bypassing freshness filters.

The federation synchronization protocol is outside the scope of this specification and will be defined in a companion document.

7. Session Establishment

Session establishment in CIRP follows a three-phase model that separates discovery, authorization, and cryptographic session setup. The critical architectural property is that the Registry participates only in the first two phases. Once authorization is complete, the Registry exits the protocol path entirely. All subsequent communication occurs on a peer-to-peer encrypted channel that the Registry can neither observe nor influence.

The three phases are:

1. Registry Discovery: The Consumer selects a Registry based on measured network latency.

2. Authorization and Ticket Issuance: The Consumer requests authorization to contact a Provider of a given capability. The Registry issues a signed ConnectTicket encoding the authorization decision.

3. Peer Session: The Consumer contacts the Provider directly, presenting the ConnectTicket. The peers negotiate a cryptographic suite, perform key exchange, and establish an encrypted channel.

This design ensures that connection throughput scales with the number of endpoints, not with Registry capacity. The Registry handles O(1) authorization per connection; it does not relay ongoing session traffic.

7.1. Registry Discovery

A Consumer MAY be aware of multiple Registries through configuration, DNS service records, or prior interaction. When multiple Registries are available, the Consumer SHOULD select the Registry with the lowest measured round-trip latency to minimize authorization delay.

Latency measurement is performed by sending a Probe message to each candidate Registry and measuring the time until a ProbeResponse is received. The Probe message contains an 8-byte nonce; the ProbeResponse echoes the nonce and MAY include the Registry's geographic locality hint. Registries MUST process Probe messages before any authentication or admission checks to ensure the measured latency reflects network conditions rather than processing overhead.

Probe messages are subject to a separate rate limit from other Registry interactions to prevent measurement traffic from interfering with authorization and discovery operations.

7.2. Authorization and Ticket Issuance

To establish a session with a Provider of a given capability, the Consumer sends an Authorization Request to a Registry. The request contains the SHA-256 hash of the desired capability URI and the Consumer's Endpoint Identifier (EID).

Upon receiving an Authorization Request, the Registry performs the following steps in order:

1. Validates that the Consumer is in an admitted state. If not, the Registry returns a NotAdmitted status and takes no further action.

2. Checks per-Consumer, per-capability rate limits. If the request exceeds the Consumer's rate allocation, the Registry returns a RateLimited status.

3. Queries its capability directory for Providers currently advertising the requested capability. Providers whose most recent presence beacon is older than a freshness threshold (measured from original beacon timestamp, not from any intermediate synchronization event) are excluded from the candidate set.

4. Applies any applicable visibility scope restrictions (see Section 6).

5. Selects a Provider from the eligible candidate set.

6. Mints a ConnectTicket binding the Consumer, the selected Provider, and the requested capability.

7. Signs the ConnectTicket with the Registry's current signing key.

8. Returns an Authorization Response containing the Provider's EID, the Provider's network locator, and the signed ConnectTicket.

7.2.1. ConnectTicket Structure

The ConnectTicket is a fixed-size, binary-encoded authorization artifact. Its structure is as follows:

ConnectTicket (272 bytes)

Offset	Size	Field
0	32	consumer_eid
32	32	consumer_vk
64	32	provider_eid
96	32	capability_hash
128	1	scope_flags
129	1	tier
130	2	rate_window_secs (BE)
132	1	rate_limit
133	8	issued_at (Unix seconds, BE)
141	8	expires_at (Unix seconds, BE)
149	16	nonce
165	8	bucket_id
173	32	issuer_eid
205	1	issuer_key_id
206	2	issuer_locality (top 16 bits)
208	64	signature (Ed25519, over bytes 0..208)

The signature covers the first 208 bytes of the ticket (all fields except the signature itself). The signature is computed using the Registry's Ed25519 signing key identified by issuer_key_id.

The ConnectTicket has the following security properties:

Identity-bound The ticket names both the Consumer and Provider by EID. Because EID equals the node's Ed25519 public key, the Consumer must prove possession of the corresponding private key when using the ticket (by signing the session initiation message). A stolen ticket is useless without the Consumer's private key.

Time-bounded The expires_at field limits the ticket's validity

window. Implementations SHOULD use a validity period of 30 seconds. Providers MUST reject tickets where the current time exceeds expires_at.

Capability-scoped The ticket authorizes contact for a specific capability only, identified by capability_hash.

Locally verifiable The Provider verifies the ticket by checking the Registry's signature using the Registry's public key. No round-trip to the Registry is required. The Provider learns the Registry's public key through control plane interactions (e.g., presence acknowledgments that carry the Registry's verifying key).

Replay-resistant The nonce field, combined with nonce-tracking at the Provider, limits reuse of a single ticket.

The issuer_key_id field supports key rotation. Registries maintain a keyring of signing keys indexed by rotation identifier. When a Provider receives a ticket, it extracts issuer_key_id and looks up the corresponding verifying key. This allows Registries to rotate signing keys without invalidating tickets issued by the previous key during an overlap window.

7.2.2. Authorization Status Codes

The Authorization Response includes a status byte indicating the outcome:

Code	Name	Description
0x00	Success	Ticket issued; Provider EID and locator included.
0x01	NoMatchingProviders	No Provider currently advertises the capability.
0x02	RateLimited	Consumer has exceeded its rate allocation.
0x03	NotAdmitted	Consumer is not in an admitted state.
0x04	PolicyBlocked	A visibility or scope policy prevents the request.

Table 2

When the status is not Success, the response MUST NOT contain a ConnectTicket. Implementations SHOULD include a policy receipt (see Section 6) with non-Success responses to support transparent diagnostics.

7.3. Cryptographic Suite Negotiation

After receiving an Authorization Response with a ConnectTicket, the Consumer initiates a direct connection to the Provider. Before exchanging key material, the peers MUST negotiate a cryptographic suite.

Suite negotiation occurs as the first exchange on the peer-to-peer channel, before any key exchange messages. This keeps the Registry out of cryptographic decisions and avoids coupling Registry upgrades to cipher suite changes.

7.3.1. Suite Identifiers

A suite identifier is a string token registered in the CIRP Crypto Suite Registry (see Section 12.1). Each suite identifier specifies the complete set of algorithms used for a session:

Key agreement The algorithm(s) used to establish a shared secret (e.g., X25519, X25519 combined with ML-KEM-768).

Authentication The signature algorithm used to authenticate key exchange messages and verify identity (e.g., Ed25519).

Symmetric encryption The authenticated encryption algorithm used for session data (e.g., ChaCha20-Poly1305).

Key derivation The key derivation function used to derive session keys from the shared secret (e.g., HKDF-SHA-256).

The following suite is mandatory to implement:

CIRP_X25519_ED25519_CHACHA20POLY1305_SHA256

Key agreement:	X25519 [RFC7748]
Authentication:	Ed25519 [RFC8032]
Symmetric encryption:	ChaCha20-Poly1305 [RFC8439]
Key derivation:	HKDF-SHA-256 [RFC5869]

The following hybrid suite is defined for post-quantum transition:

CIRP_X25519MLKEM768_ED25519_CHACHA20POLY1305_SHA256

Key agreement:	X25519 [RFC7748] combined with ML-KEM-768 [FIPS203]
Authentication:	Ed25519 [RFC8032]
Symmetric encryption:	ChaCha20-Poly1305 [RFC8439]
Key derivation:	HKDF-SHA-256 [RFC5869]

7.3.2. Negotiation Protocol

Suite negotiation consists of two messages:

SuiteOffer (Consumer to Provider) Contains the Consumer's ConnectTicket, the Consumer's session identifier, and an ordered list of supported suite identifiers. The first entry is the Consumer's most preferred suite. The Consumer MUST sign the SuiteOffer with its Ed25519 private key to prove ownership of the EID named in the ticket.

SuiteSelect (Provider to Consumer) Contains the Provider's selected suite identifier (a single value). The Provider MUST sign the SuiteSelect with its Ed25519 private key.

Upon receiving a SuiteOffer, the Provider:

1. Validates the ConnectTicket (verifies the Registry's signature, checks expiration, confirms the Provider's own EID matches the ticket's provider_eid, and verifies the Consumer's signature on the SuiteOffer against the consumer_vk in the ticket).

2. Selects the first suite from the Consumer's ordered list that the Provider also supports.

3. If no common suite exists, the Provider silently drops the connection. The Consumer will observe a timeout.

4. Returns a signed SuiteSelect containing the chosen suite identifier.

Upon receiving a SuiteSelect, the Consumer:

1. Verifies the Provider's signature.

2. Confirms the selected suite was present in the Consumer's original SuiteOffer. If the selected suite was NOT offered, the Consumer MUST abort the connection. This prevents downgrade attacks where a network attacker substitutes the SuiteSelect message.

3. Proceeds to key exchange using the negotiated suite.

7.4. Key Exchange

After suite negotiation, the peers perform a key exchange to establish shared session keys. The key exchange protocol depends on the negotiated suite but follows a uniform structure.

7.4.1. Classical Key Exchange

For suites using X25519 as the sole key agreement algorithm, the key exchange proceeds as follows:

1. Each peer generates an ephemeral X25519 keypair for this session.

2. Each peer sends a KeyExchange message containing: the session identifier, a role indicator (0x01 for Consumer, 0x02 for Provider), its ephemeral X25519 public key (32 bytes), and an Ed25519 signature over these fields using the peer's long-term identity key.

3. Each peer performs X25519 Diffie-Hellman using its ephemeral private key and the other peer's ephemeral public key, producing a 32-byte shared secret.

4. Session keys are derived from the shared secret using the key schedule defined in Section 7.4.3.

The Ed25519 signatures on the KeyExchange messages bind the ephemeral keys to the peers' long-term identities. An attacker who intercepts the ephemeral public keys cannot forge the signatures without possessing the peers' long-term private keys.

Ephemeral keys MUST be generated fresh for each session and MUST NOT be reused. This provides forward secrecy: compromise of a peer's long-term Ed25519 signing key does not expose the session keys of previously established sessions.

7.4.2. Hybrid Post-Quantum Key Exchange

For suites combining X25519 with a post-quantum key encapsulation mechanism (such as ML-KEM-768 as specified in [FIPS203]), the key exchange produces two independent shared secrets that are combined via the key schedule.

The hybrid exchange proceeds as follows:

1. The Consumer generates an ephemeral X25519 keypair and an ephemeral ML-KEM-768 keypair. The Consumer sends both ephemeral public keys in its KeyExchange message, signed with its Ed25519 identity key.

2. The Provider generates an ephemeral X25519 keypair. The Provider performs X25519 Diffie-Hellman to obtain the classical shared secret. The Provider encapsulates against the Consumer's ML-KEM-768 public key to obtain a PQ ciphertext and the PQ shared secret. The Provider sends its X25519 ephemeral public key and the ML-KEM-768 ciphertext, signed with its Ed25519 identity key.

3. The Consumer performs X25519 Diffie-Hellman to obtain the classical shared secret. The Consumer decapsulates the ML-KEM-768 ciphertext to obtain the PQ shared secret.

4. Both peers now hold two shared secrets: classical_ss (from X25519) and pq_ss (from ML-KEM-768). Session keys are derived using the key schedule in Section 7.4.3.

When operating in hybrid mode, both the classical and post-quantum key exchanges MUST succeed. If either exchange fails, session establishment MUST fail. An implementation MUST NOT fall back to classical-only key agreement when a hybrid suite has been negotiated. This prevents downgrade to classical-only security when both peers have agreed to hybrid protection.

7.4.3. Key Schedule

Regardless of the negotiated suite, session keys are derived using HKDF [RFC5869] with SHA-256 as the hash function, following the extract-then-expand paradigm of [NIST-SP-800-56C]:

```
For classical suites:
  ikm = classical_ss

For hybrid suites:
  ikm = classical_ss || pq_ss

PRK   = HKDF-Extract(salt=session_id, ikm)
key   = HKDF-Expand(PRK, info, L=32)

where:
  session_id = 16-byte session identifier
  info       = "cirp-hybrid-kx" || suite_id ||
               consumer_eid || provider_eid
  L          = 32 (256 bits, for ChaCha20-Poly1305)
```

The info string binds the derived key material to the session context and both peer identities. The suite_id is the ASCII-encoded suite identifier string as negotiated. The consumer_eid and provider_eid are the 32-byte EIDs of the respective peers.

Using the same KDF structure for both classical and hybrid suites simplifies implementation and allows the key derivation path to be verified independently of the key agreement mechanism.

7.5. Encrypted Session

Once session keys are derived, the peers communicate using authenticated encrypted frames. Each frame uses ChaCha20-Poly1305 [RFC8439] for authenticated encryption with associated data (AEAD).

The encrypted frame format is:

Encrypted Frame

Offset	Size	Field
0	4	magic (0x41 0x49 0x43 0x46)
4	16	session_id
20	8	counter (monotonic, BE)
28	12	nonce (ChaCha20-Poly1305)
40	N	ciphertext
40+N	16	authentication_tag (Poly1305)

Minimum frame size: 56 bytes (empty payload)

The counter field is a monotonically increasing 64-bit integer, incremented for each frame sent within a session. Receivers MUST reject frames with a counter value less than or equal to the highest counter previously accepted in the same session. This provides replay protection without requiring synchronized state beyond the highest seen counter.

The nonce is constructed deterministically from the counter and session key material. Implementations MUST NOT reuse a nonce with the same key.

The protocol imposes no semantic limit on payload size within encrypted frames. Implementations MAY enforce practical limits based on the underlying transport binding. For UDP transport, implementations typically target application payloads that fit within common path MTU limits and handle larger payloads through fragmentation and reassembly at the session layer.

The ciphertext is opaque to the transport layer. It carries application data (including intent invocation envelopes as defined in Section 9) without interpretation, canonicalization, or transformation by the transport.

8. Message Framing and Encapsulation

CIRP uses distinct framing for control plane and data plane messages to enable demultiplexing on shared transport sockets.

8.1. Control Plane Framing

Control plane messages between endpoints and Registries use a type-length-value (TLV) envelope:

Control Plane TLV Envelope

Offset	Size	Field
0	1	message_type
1	2	payload_length (BE)
3	var	payload

The message_type field identifies the control plane operation (e.g., presence announcement, discovery query, authorization request). The payload_length field is a 16-bit big-endian unsigned integer specifying the number of bytes following the header.

8.2. Data Plane Framing

Data plane messages between peers use magic-byte prefixed frames. Two frame types are defined:

Key Exchange Frame (magic: 0x41 0x49 0x4B 0x58, ASCII "AIKX")
: Carries suite negotiation, ephemeral public keys, and signatures during session establishment (Section 7.4). Fixed structure: magic (4 bytes) + session_id (16) + role (1) + ephemeral_pk (32) + signature (64) = 117 bytes for classical suites. Hybrid suites include additional key material.

Encrypted Frame (magic: 0x41 0x49 0x43 0x46, ASCII "AICF")
: Carries encrypted application data including invocation and fulfillment envelopes. Structure defined in Section 7.5: magic (4) + session_id (16) + counter (8) + nonce (12) + ciphertext (N) + tag (16). Minimum 56 bytes.

Receivers distinguish control plane from data plane messages by inspecting the first byte: TLV type values occupy a different range from the ASCII 'A' (0x41) that begins both data plane magic sequences.

8.3. Framing Properties

The framing layer provides the following guarantees:

Payload opacity The framing layer does not inspect, parse, or transform the contents of encrypted frames. Ciphertext is carried as opaque bytes.

Byte safety No canonicalization, character encoding conversion, or normalization is applied to payload data at any layer of the protocol. Binary payloads are preserved exactly as submitted.

Replay protection Encrypted frames carry a monotonically increasing counter. Receivers reject frames with counter values not greater than the highest previously accepted value in the same session.

9. Intent Invocation

Intent invocation is the mechanism by which a Consumer requests execution of a capability and receives a fulfillment response from a Provider. Invocation messages are carried within the encrypted peer-to-peer session established in Section 7. The transport layer does not interpret, validate, or transform the payload contents of invocations or fulfillments.

Invocation envelopes are encoded using CBOR [RFC8949] with deterministic encoding as specified in Section 4.2 of that document. Deterministic encoding ensures that the same logical content always produces identical byte sequences, which is essential for stable signatures and hash computation. CBOR map keys are encoded as unsigned integers for compactness.

9.1. Request Envelope

The invocation request envelope contains the following fields, encoded as a CBOR map with integer keys:

Key	Field	CBOR Type	Size	Description
1	invocation_id	bstr	16	Unique identifier for this

					invocation, generated randomly by the Consumer.
2	capability_uri	tstr	variable	Full capability URI including version (e.g., "cap:robot.wave/v1.0"). This is the human-readable form; the hash used during discovery can be recomputed from this value.
3	payload_type	tstr	variable	Identifier for the payload encoding. This MAY be a MIME type or an application-defined type string. The protocol does not interpret this value.
4	payload	bstr	variable	Opaque payload bytes. The protocol does not parse, canonicalize, or transform this field. Content interpretation is entirely the responsibility of the Consumer and Provider.
5	consumer_eid	bstr	32	Consumer’s Endpoint Identifier.
6	consumer_send_ts	uint	8	Consumer’s local send timestamp, unsigned 64-bit milliseconds since Unix epoch.
7	prev_invocation_hash	bstr	32	SHA-256 hash of the

					previous invocation envelope in this Consumer-Provider relationship. Set to 32 zero bytes for the first invocation. See Section 10.3.
8	consumer_signature	bstr	64		Ed25519 signature computed over the deterministic CBOR encoding of fields 1 through 7 (all fields except the signature itself).

Table 3

The Consumer MUST generate the invocation_id using a cryptographically secure random number generator. Invocation identifiers are used for correlation between requests, responses, error frames, and fulfillment records.

The signature is computed over the canonical CBOR encoding of a map containing keys 1 through 7 with their values. The Consumer uses its Ed25519 signing key (the private key corresponding to consumer_eid) to produce the signature. This binds the invocation content to the Consumer’s identity and prevents tampering in transit.

9.2. Response Envelope

The Provider responds to an invocation with a fulfillment response encoded as a CBOR map:

Key	Field	CBOR Type	Size	Description
1	invocation_id	bstr	16	Copied from the request for correlation.
2	fulfillment_status	uint	1	Status of the fulfillment: 0x00 for success, 0x01 for partial, 0x02

				for application error (details in payload).
3	payload_type	tstr	variable	Identifier for the response payload encoding.
4	payload	bstr	variable	Opaque response payload. Not parsed by the protocol.
5	provider_eid	bstr	32	Provider’s Endpoint Identifier.
6	provider_recv_ts	uint	8	Provider’s local timestamp when the request was received.
7	provider_send_ts	uint	8	Provider’s local timestamp when the response was sent.
8	request_hash	bstr	32	SHA-256 hash of the complete request envelope (all bytes including the Consumer’s signature). Binds the response to a specific request.
9	provider_signature	bstr	64	Ed25519 signature over the deterministic CBOR encoding of fields 1 through 8.

Table 4

The Provider MUST include the request_hash computed over the entire request envelope as received. This cryptographically binds the response to the specific request, preventing a malicious party from associating a valid response with a different request.

When the fulfillment_status is 0x02 (application error), the payload field carries application-specific error information. The protocol does not define the structure of application-level errors; they are opaque bytes interpreted solely by the Consumer and Provider.

9.3. Error Semantics

CIRP distinguishes between protocol-level errors and application-level errors. Application-level errors are carried inside the response payload (fulfillment_status 0x02) and are opaque to the protocol. Protocol-level errors indicate failures in session establishment, authorization, or transport that prevent an invocation from reaching the Provider or a response from reaching the Consumer.

Protocol-level errors are conveyed in signed error frames. Signing error frames prevents an active network attacker from injecting spurious errors to disrupt communication.

An error frame is encoded as a CBOR map:

Key	Field	CBOR Type	Description
1	invocation_id	bstr	Correlation identifier from the request, if available. Set to 16 zero bytes if the error is not associated with a specific invocation.
2	error_code	uint	Numeric error code from the CIRP Error Code registry.
3	error_detail	tstr	Optional human-readable description. Implementations MUST NOT rely on the content of this field for programmatic error handling.
4	error_origin	uint	Indicates which component generated the error: 0x01 for Registry (authorization phase), 0x02 for Provider (invocation phase), 0x03 for transport (timeout or connection failure).
5	originator_eid	bstr	EID of the entity that generated the error.
6	signature	bstr	Ed25519 signature over fields 1 through 5.

Table 5

9.3.1. Protocol Error Codes

The following error codes are defined. Additional codes may be registered in the CIRP Protocol Parameters Registry (Section 12.3).

Code	Name	Description
0x01	CAPABILITY_NOT_FOUND	No Provider advertises the requested capability.
0x02	PROVIDER_UNAVAILABLE	The Provider is not reachable or has become unresponsive.
0x03	AUTHORIZATION_EXPIRED	The ConnectTicket has expired.
0x04	TICKET_INVALID	The ConnectTicket failed validation.
0x05	SUITE_MISMATCH	No common cryptographic suite exists between the peers.
0x06	RATE_LIMITED	The request exceeds the Consumer’s rate allocation.
0x07	SCOPE_DENIED	The Consumer’s identity does not satisfy the capability’s scope requirements.
0x08	TIMEOUT	The operation exceeded the applicable time limit.
0x09	INTERNAL_ERROR	An unspecified internal error occurred.

Table 6

10. Receipts and Integrity

CIRP provides a mutually attested record of each invocation-fulfillment exchange. This record, called a fulfillment receipt, is constructed cooperatively by the Provider and Consumer through a two-phase signing ceremony. The receipt establishes a tamper-evident, non-repudiable audit trail without relying on a trusted third party or synchronized clocks.

10.1. Fulfillment Record Structure

The fulfillment record is encoded as a CBOR map using deterministic encoding ([RFC8949], Section 4.2). The record is constructed in two phases:

10.1.1. Phase 1: Provider Attestation

After processing an invocation and generating a response, the Provider constructs a partial fulfillment record containing the following fields:

Key	Field	CBOR Type	Description
1	invocation_id	bstr (16)	Identifier of the invocation this receipt covers.
2	request_hash	bstr (32)	SHA-256 hash of the complete request envelope as received.
3	response_hash	bstr (32)	SHA-256 hash of the complete response envelope as sent.
4	provider_recv_ts	uint	Provider’s local clock reading when the request was received (milliseconds since epoch).
5	provider_send_ts	uint	Provider’s local clock reading when the response was sent.
6	provider_eid	bstr (32)	Provider’s Endpoint Identifier.
7	provider_signature	bstr (64)	Ed25519 signature over the deterministic CBOR encoding of fields 1 through 6.

Table 7

The Provider sends this partial record alongside the response envelope. The Provider’s signature attests that the Provider processed the identified request and produced the identified response at the stated times.

10.1.2. Phase 2: Consumer Finalization

Upon receiving the response and the Provider’s partial record, the Consumer appends additional fields and produces the final fulfillment record:

Key	Field	CBOR Type	Description
1-7	(Provider fields)	(as above)	All fields from Phase 1 including the Provider's signature.
8	consumer_send_ts	uint	Consumer's local clock reading when the request was sent.
9	consumer_recv_ts	uint	Consumer's local clock reading when the response was received.
10	consumer_eid	bstr (32)	Consumer's Endpoint Identifier.
11	consumer_signature	bstr (64)	Ed25519 signature over the deterministic CBOR encoding of fields 1 through 10 (including the Provider's signature).

Table 8

The Consumer's signature covers the entire record including the Provider's signature at field 7. This creates a nested attestation: the Consumer attests that it received the Provider's signed partial record and that the Consumer's own timestamps are accurate. Neither party can tamper with the other's attested fields without invalidating the corresponding signature.

10.1.3. Receipt Verification

A third party verifying a fulfillment record performs the following checks:

1. Reconstruct the CBOR map of fields 1 through 6 using deterministic encoding. Verify the Provider's signature (field 7) against this encoding using the provider_eid as the verification key.

2. Reconstruct the CBOR map of fields 1 through 10 using deterministic encoding. Verify the Consumer's signature (field 11) against this encoding using the consumer_eid as the verification key.

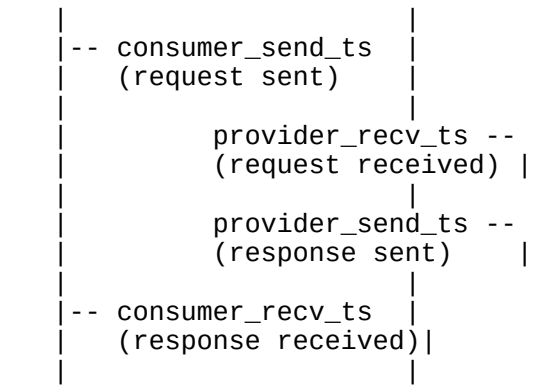
3. Confirm that the provider_eid and consumer_eid match the expected parties.

4. Optionally verify that request_hash and response_hash match the actual request and response envelopes if those are available.

If both signatures verify, the receipt cryptographically establishes that both parties participated in the exchange and attested to its contents.

10.2. Timestamp Model

The fulfillment record contains four timestamps representing the complete round-trip of an invocation:



Timestamps are encoded as unsigned 64-bit integers representing milliseconds since the Unix epoch (1970-01-01T00:00:00Z).

Clock synchronization between Consumer and Provider is NOT assumed. Each party records timestamps using its own local clock. The timestamp model is designed for relative timing analysis rather than absolute time agreement:

Provider processing time provider_send_ts - provider_recv_ts (single clock, reliable).

Observed round-trip time consumer_recv_ts - consumer_send_ts (single clock, reliable).

Estimated one-way latency (consumer_recv_ts - consumer_send_ts) - (provider_send_ts - provider_recv_ts), divided by two. This estimate requires no clock synchronization because the subtracted intervals are each measured on a single clock.

The consumer_send_ts and provider_send_ts fields are REQUIRED (MUST be present). The provider_recv_ts and consumer_recv_ts fields SHOULD be present; omitting them degrades the usefulness of the receipt for latency analysis but does not invalidate the record.

Implementations MUST NOT reject receipts solely because timestamps appear inconsistent (e.g., provider_recv_ts earlier than consumer_send_ts). Such inconsistency indicates clock skew, not necessarily fraud. Implementations MAY flag such records for review.

10.3. Hash Chaining

Each invocation request envelope includes a prev_invocation_hash field (see Section 9.1) that references the SHA-256 hash of the immediately preceding request envelope in the same Consumer-Provider relationship. For the first invocation between a given Consumer and Provider, this field is set to 32 zero bytes.

This creates a hash chain that provides the following properties:

Ordering evidence Each invocation cryptographically references its predecessor, establishing an order that cannot be rearranged without detection.

Deletion detection Removing an invocation from the chain breaks the hash reference in the subsequent invocation, making omission detectable.

Per-relationship isolation The chain is scoped to a specific Consumer-Provider pair. Invocations between different pairs maintain independent chains.

Hash chains are maintained by the Consumer. A Consumer MUST track the hash of its most recent request envelope for each active Provider relationship. If a Consumer loses this state (e.g., due to a restart), it SHOULD set prev_invocation_hash to 32 zero bytes, effectively starting a new chain. The Provider MAY note chain resets for audit purposes but MUST NOT reject invocations solely because the hash chain was reset.

11. Security Considerations

This section analyzes the security properties of CIRP under a threat model that assumes an active network attacker, potentially compromised Registries, and potentially compromised peer endpoints. The analysis addresses each threat class and identifies the cryptographic mechanisms that mitigate it.

11.1. Threat Model

The protocol is designed to resist the following adversaries:

Active network attacker An adversary positioned on the network path who can observe, inject, modify, delay, replay, and drop messages. This is the standard Dolev-Yao network attacker model.

Compromised Registry A Registry that deviates from protocol specification, either through compromise or malicious operation. The analysis bounds what such a Registry can achieve.

Compromised peer An endpoint whose long-term signing key has been compromised. The analysis addresses the impact on past and future sessions.

Harvest-now-decrypt-later attacker An adversary who records encrypted traffic today with the expectation of decrypting it in the future using advances in cryptanalysis or quantum computing.

11.2. Mutual Authentication

Every endpoint in CIRP is identified by its Ed25519 [RFC8032] public key. The Endpoint Identifier (EID) is the public key itself, not a hash or derivative. This identity model has several consequences:

* No certificate authority is required. Identity is self-asserted and verified by checking signatures against the claimed EID.

* Impersonation requires possession of the victim’s Ed25519 private key. There is no weaker path (such as compromising a CA) to forge an identity.

* All protocol messages that assert identity (ConnectTickets, SuiteOffer, SuiteSelect, KeyExchange, invocation envelopes, response envelopes, error frames, and fulfillment records) are signed by the originator’s Ed25519 key.

During session establishment, mutual authentication is achieved through the following chain: the Consumer proves identity by signing the SuiteOffer (which includes the ConnectTicket naming the Consumer's EID), and the Provider proves identity by signing the SuiteSelect and KeyExchange messages using the key corresponding to the EID named in the ticket's provider_eid field. Both parties verify these signatures before proceeding to key exchange.

11.3. Registry Trust Boundary

Registries occupy a semi-trusted role. The following analysis enumerates what a Registry can and cannot do, and provides cryptographic justification for each boundary.

11.3.1. Actions a Registry Can Perform

Deny service A Registry can refuse to process discovery queries or authorization requests, preventing new sessions from being established through that Registry. Mitigation: endpoints can use alternative Registries.

Filter discovery results A Registry can omit Providers from discovery responses, effectively hiding capabilities. Mitigation: Policy Receipts make filtering observable; multi-Registry deployments provide independent views.

Refuse ticket issuance A Registry can decline to issue ConnectTickets, blocking specific Consumer-Provider pairs. Mitigation: same as denial of service.

Shape visibility via scoping A Registry enforces scope restrictions during discovery. A malicious Registry could misapply scope rules to restrict or expand visibility beyond the Provider's intent.

Observe authorization metadata A Registry necessarily observes which Consumer requested authorization to contact which Provider for which capability. This metadata is visible to the Registry by design.

11.3.2. Actions a Registry Cannot Perform

Forge endpoint identities Because EID equals the Ed25519 public key and there is no certificate authority, a Registry cannot create a valid identity for an endpoint it does not control. Signing a message as a given EID requires the corresponding private key, which the Registry does not possess.

Decrypt data plane traffic Session keys are derived from ephemeral

X25519 [RFC7748] Diffie-Hellman exchanges performed directly between peers. The Registry is not a party to the key exchange and does not receive ephemeral key material. Without the shared secret, the Registry cannot derive session keys or decrypt any data plane frame.

Observe invocation content Invocation envelopes, fulfillment responses, and application payloads are carried exclusively on the data plane within encrypted frames. The Registry is architecturally excluded from the data plane (Section 4.2). It never receives, relays, or processes data plane messages.

Tamper with invocation records Fulfillment records are dual-signed: the Provider signs fields 1 through 6, and the Consumer signs fields 1 through 10 including the Provider's signature (Section 10.1). Modifying any field invalidates the corresponding signature. Since the Registry possesses neither party's signing key, it cannot produce valid replacement signatures.

Issue tickets for endpoints it does not serve A ConnectTicket is signed by the issuing Registry's key. A Provider verifies the ticket signature against the Registry's known public key. A Registry cannot issue tickets that will be accepted by Providers that trust a different Registry, unless it possesses that other Registry's signing key.

11.4. Ticket Security

The ConnectTicket (Section 7.2.1) is the authorization boundary between the control plane and the data plane. Its security depends on several properties:

Identity binding The ticket names both Consumer and Provider by EID. The Consumer must sign the SuiteOffer using the private key corresponding to the consumer_eid in the ticket. A stolen ticket is useless without the Consumer's private key.

Time bounding The ticket carries an expires_at field. Implementations SHOULD use a 30-second validity window. Providers MUST reject expired tickets. Short validity limits the window for ticket theft and replay.

Nonce-based replay resistance The ticket contains a 16-byte nonce. Providers track seen nonces and reject tickets with previously used nonces. Combined with time bounding, this limits replay to the narrow validity window.

Capability scoping The ticket binds authorization to a specific

capability hash. A ticket issued for one capability cannot be used to invoke a different capability.

Local verifiability Providers verify the Registry's signature on the ticket using the Registry's known public key. No round-trip to the Registry is needed. This means ticket verification succeeds even if the Registry becomes temporarily unreachable after ticket issuance.

11.5. Forward Secrecy

Each session uses freshly generated ephemeral X25519 keypairs for key exchange. Session keys are derived from the ephemeral shared secret, not from the peers' long-term Ed25519 keys. Compromise of a peer's long-term signing key allows an attacker to impersonate that peer in future sessions, but it does not expose the session keys of previously established sessions because the ephemeral X25519 private keys are discarded after key derivation.

Long-term Ed25519 keys are used solely for authentication (signing key exchange messages and protocol artifacts), not for key agreement. The key agreement function (X25519) operates on independent ephemeral key material.

11.6. Replay Protection

Replay attacks are addressed at multiple layers:

Encrypted frames Each encrypted frame carries a monotonically increasing counter (Section 7.5). Receivers reject frames with counter values less than or equal to the highest previously accepted counter in the session. Because the counter is included in the authenticated encryption, an attacker cannot modify it without detection.

ConnectTickets Tickets are time-bounded (expires_at) and nonce-tracked at the Provider. Replaying an expired ticket or reusing a seen nonce results in rejection.

Key exchange messages Each KeyExchange message is bound to a specific session_id and signed by the sender's long-term key. Replaying a KeyExchange message from a previous session in a new session produces a mismatched session_id, causing verification failure.

11.7. Downgrade Protection

Two mechanisms prevent cryptographic downgrade attacks:

Suite selection validation The Consumer MUST abort the connection if the Provider selects a suite that was not present in the Consumer's SuiteOffer (Section 7.3.2). This prevents an active attacker from substituting the SuiteSelect message to force a weaker suite. Both the SuiteOffer and SuiteSelect are signed, so the attacker cannot modify them without detection.

Hybrid exchange integrity When a hybrid post-quantum suite is negotiated, both the classical and post-quantum key exchanges MUST succeed (Section 7.4.2). If either fails, session establishment fails entirely. An attacker cannot force fallback to classical-only key agreement after both peers have agreed to hybrid protection.

11.8. Post-Quantum Transition

CIRP addresses the threat of harvest-now-decrypt-later attacks through its cryptographic agility framework and hybrid key exchange construction.

The protocol supports hybrid post-quantum key establishment via negotiated cipher suites. In hybrid mode, session keys are derived from both a classical X25519 shared secret and a post-quantum shared secret (e.g., from ML-KEM-768 as specified in [FIPS203]), combined through HKDF (Section 7.4.3). This construction ensures that the session key is at least as strong as the stronger of the two components: even if the post-quantum algorithm is later found to be weak, the classical component still provides security, and vice versa.

The suite negotiation mechanism (Section 7.3) allows new cryptographic suites to be deployed without protocol revision. As post-quantum algorithms mature and new key encapsulation mechanisms are standardized, they can be registered in the CIRP Crypto Suite Registry and adopted by implementations through normal suite negotiation.

Implementations MUST support suite negotiation as defined in Section 7.3. Implementations MUST support at least the mandatory classical suite. Implementations SHOULD support at least one hybrid post-quantum suite. Future revisions of this specification may strengthen the hybrid requirement to MUST as post-quantum algorithms achieve broader deployment maturity.

11.9. Invocation Integrity

Invocation envelopes and response envelopes are signed by the originating party using Ed25519. The signature covers the deterministic CBOR encoding of all fields except the signature itself (Section 9.1, Section 9.2). This provides:

* Authentication: the envelope was produced by the claimed originator.

* Integrity: the envelope has not been modified since signing.

* Non-repudiation: the originator cannot deny having produced the envelope without claiming key compromise.

The response envelope includes a request_hash field that binds the response to a specific request. This prevents an attacker from associating a legitimate response with a different request.

Fulfillment records extend these properties through cooperative dual-signing (Section 10.1), creating a mutually attested audit trail that neither party can unilaterally repudiate.

11.10. Error Frame Security

Protocol-level error frames are signed by the originating entity (Section 9.3). This prevents an active network attacker from injecting spurious error messages to disrupt communication. A receiver MUST verify the signature on an error frame before acting on it. Unsigned or incorrectly signed error frames MUST be discarded.

11.11. Scope Enforcement Security

Capability visibility scoping (Section 6.3) is enforced by the Registry during discovery. Because scope enforcement depends on the Registry correctly evaluating scope rules, a compromised Registry could bypass scope restrictions and expose capabilities intended to be private.

Deployments with strict confidentiality requirements for capability visibility SHOULD use dedicated Registries operated within the trust domain. Capability scoping provides defense in depth but does not replace network-level access controls where confidentiality of capability existence is critical.

11.12. Denial of Service Considerations

Several protocol elements are susceptible to denial of service attacks:

Discovery flooding An attacker may send a high volume of discovery queries to exhaust Registry resources. Registries SHOULD implement per-source rate limiting on discovery queries. The admission control mechanism provides a first line of defense: unadmitted endpoints can be rejected before query processing begins.

Ticket exhaustion An attacker may request ConnectTickets at high rate to exhaust Provider nonce tracking state. Per-consumer, per-capability rate limiting at the Registry bounds the rate at which tickets are issued. The short ticket validity window (30 seconds) limits the volume of unexpired tickets in circulation.

Probe flooding An attacker who obtains valid ConnectTickets may flood a Provider with connection probes. Providers SHOULD implement per-source connection rate limits. The four-phase validation order (structural, cryptographic, semantic, replay) ensures that invalid probes are rejected with minimal processing, with cryptographically invalid probes silently dropped to prevent oracle attacks.

11.13. Time Skew Considerations

The protocol relies on timestamp comparison for ConnectTicket expiration checking. Clock skew between the Registry (which sets issued_at and expires_at) and the Provider (which checks expiration) may cause valid tickets to be incorrectly rejected or expired tickets to be incorrectly accepted.

Implementations SHOULD allow a small clock skew tolerance (on the order of seconds) when checking ticket expiration. The 30-second recommended validity window provides margin for modest clock drift. Deployments operating across geographically distributed infrastructure SHOULD ensure that Registry and Provider clocks are synchronized to within a few seconds using NTP or equivalent mechanisms.

Fulfillment record timestamps (Section 10.2) explicitly do not assume clock synchronization between Consumer and Provider. Each party records timestamps on its own clock. The timestamp model is designed for single-clock interval analysis (processing time, round-trip time) rather than cross-clock absolute time comparisons.

12. IANA Considerations

This document requests the creation of the following registries upon publication.

12.1. CIRP Crypto Suite Registry

IANA is requested to create a "CIRP Crypto Suite Registry" with the following initial entries. New entries require Specification Required registration policy.

Suite Identifier	Status
CIRP_X25519_ED25519_CHACHA20POLY1305_SHA256	Mandatory
CIRP_X25519MLKEM768_ED25519_CHACHA20POLY1305_SHA256	Recommended

Table 9

Each suite identifier encodes its component algorithms in the name: key agreement, authentication, AEAD, and KDF, separated by underscores. The algorithms for each initial entry are specified in Section 7.3.1.

Suite identifiers are ASCII strings. The naming convention concatenates the key agreement, authentication, AEAD, and KDF algorithm names separated by underscores, prefixed with "CIRP_".

The ML-KEM-768 algorithm in the hybrid suite is as specified in [FIPS203].

12.2. URI Scheme Registration

IANA is requested to register the "cap" URI scheme in the "Uniform Resource Identifier (URI) Schemes" registry per [RFC7595].

Scheme name cap

Status Permanent

Applications/protocols that use this scheme CIRP (Capability-Oriented Intent Routing Protocol)

Contact IESG

Change controller IETF

Reference This document, Section 5.1

12.3. CIRP Protocol Parameters Registry

IANA is requested to create a "CIRP Protocol Parameters" registry with the following sub-registries. New entries in each sub-registry require Specification Required registration policy.

12.3.1. Error Codes

Initial entries are as defined in Section 9.3.1. The registry contains: code (uint8), name (string), and description (string).

12.3.2. Scope Types

Initial entries are the five scope levels defined in Section 6.3.1. The registry contains: value (uint8), name (string), qualifier size (uint8), and description (string).

12.3.3. Message Types

This sub-registry tracks control plane TLV message types and data plane magic-byte prefixes. Initial entries include the message types referenced in Section 8.

13. References

13.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, June 2018, <https://www.rfc-editor.org/info/rfc8439>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[RFC7595] Thaler, D., Hansen, T., and T. Hardie, "Guidelines and Registration Procedures for URI Schemes", BCP 35, RFC 7595, June 2015, <https://www.rfc-editor.org/info/rfc7595>.

13.2. Informative References

[FIPS203] NIST, "Module-Lattice-Based Key-Encapsulation Mechanism Standard", FIPS 203, August 2024.

[NIST-SP-800-56C] NIST, "Recommendation for Key-Derivation Methods in Key-Establishment Schemes", SP 800-56C Rev. 2, August 2020, <https://csrc.nist.gov/publications/detail/sp/800-56c/rev-2/final>.

Examples

Example 1: Cross-Domain Intent Routing (Robotics)

This example demonstrates an invocation of a robotics capability using CIRP. A Consumer instructs a robotic arm to perform a wave gesture.

The capability URI is cap:robot.wave/v1.0. The capability hash is SHA-256("cap:robot.wave/v1.0"), yielding a 32-byte value used during discovery and authorization.

Step 1: Discovery and Authorization. The Consumer sends an Authorization Request containing the capability hash and its EID. The Registry locates a Provider advertising cap:robot.wave/v1.0, mints a ConnectTicket (272 bytes) binding the Consumer, Provider, and capability, and returns the ticket with the Provider’s network locator.

Step 2: Suite Negotiation. The Consumer sends a SuiteOffer to the Provider containing the ConnectTicket, a session_id, and offered suites [CIRP_X25519MLKEM768_ED25519_CHACHA20POLY1305_SHA256, CIRP_X25519_ED25519_CHACHA20POLY1305_SHA256], signed with the Consumer's Ed25519 key. The Provider validates the ticket, selects the hybrid suite, and returns a signed SuiteSelect.

Step 3: Key Exchange. The Consumer sends a KeyExchange message containing its ephemeral X25519 public key (32 bytes) and its ephemeral ML-KEM-768 encapsulation key, signed with its identity key. The Provider performs X25519 DH and ML-KEM-768 encapsulation, returning its ephemeral X25519 public key and the ML-KEM-768 ciphertext, signed. Both derive the session key via HKDF.

Step 4: Invocation. The Consumer constructs a request envelope:

Request Envelope (CBOR map, integer keys):

```
{
  1: h'A3B4...C5D6'            // invocation_id (16 bytes)
  2: "cap:robot.wave/v1.0"    // capability_uri
  3: "application/json"       // payload_type
  4: h'7B22...'               // payload: {"gesture":"wave",
                               //   "amplitude":0.8,"cycles":3}
  5: h'1234...5678'            // consumer_eid (32 bytes)
  6: 1708012800000             // consumer_send_ts
  7: h'0000...0000'            // prev_invocation_hash (first)
  8: h'9ABC...DEF0'            // consumer_signature (64 bytes)
}
```

The envelope is CBOR-encoded and sent within an encrypted AICF frame. The payload contains a JSON-encoded robotics command. The transport does not parse or interpret the JSON; it is carried as opaque bytes.

Step 5: Fulfillment. The Provider processes the command, actuates the robotic arm, and returns a response envelope:

```
Response Envelope (CBOR map):
{
  1: h'A3B4...C5D6'           // invocation_id (correlation)
  2: 0                        // fulfillment_status: success
  3: "application/json"      // payload_type
  4: h'7B22...'              // payload: {"status":"completed",
                              //   "duration_ms":1247}
  5: h'ABCD...EF01'           // provider_eid (32 bytes)
  6: 1708012800050            // provider_recv_ts
  7: 1708012801297            // provider_send_ts
  8: h'F0E1...D2C3'           // request_hash (32 bytes)
  9: h'5678...9ABC'           // provider_signature (64 bytes)
}
```

Step 6: Receipt. The Provider constructs Phase 1 of the fulfillment record (fields 1-7, Provider-signed) and sends it alongside the response. The Consumer appends fields 8-11 (consumer_send_ts, consumer_recv_ts, consumer_eid, consumer_signature) to produce the final dual-signed receipt.

This example demonstrates payload neutrality: the robotics command could equally be a binary protocol buffer, a domain-specific language instruction, or any other byte sequence. The transport handles all payloads identically.

Example 2: Organization-Scoped Discovery

This example demonstrates capability scoping within an organizational boundary.

An organization operates an internal document analysis service at cap:acme.docs.analyze/v1.0. The Provider advertises this capability with Organization scope:

```
Scope Descriptor (33 bytes):
  scope_level:  0x03  (Organization)
  anchor_hash:  SHA-256(organization_signing_key)
                = h'7F8E...1A2B' (32 bytes)
```

When a Consumer queries for cap:acme.docs.analyze/v1.0, the Registry evaluates the scope: it checks whether the Consumer's identity chain includes the trust anchor identified by anchor_hash. If the Consumer is a member of the organization (its EID is bound to the organization's trust anchor through a deployment-specific membership mechanism), the capability appears in discovery results. If not, the Registry excludes the capability and returns a Policy Receipt indicating scope denial.

An external Consumer querying the same Registry sees no evidence that cap:acme.docs.analyze/v1.0 exists. The scope enforcement occurs at the Registry during query processing, not at the Consumer or Provider.

Example 3: Cryptographic Suite Negotiation

This example shows the complete message-level flow of suite negotiation and hybrid key exchange between a Consumer (C) and Provider (P).

Message 1: SuiteOffer (C -> P)

```
+-----------------------------------------------+
| ConnectTicket (272 bytes)                     |
| session_id: h'01020304...10111213' (16 bytes)|
| offered_suites: [                             |
|   "CIRP_X25519MLKEM768_ED25519_CHACHA20P...",|
|   "CIRP_X25519_ED25519_CHACHA20POLY1305_..." |
| ]                                             |
| consumer_signature: Ed25519(consumer_sk,      |
|   SHA-256(ticket || session_id || suites))    |
+-----------------------------------------------+
```

```
  Provider validates:
    1. Ticket signature (Registry's VK)      -> OK
    2. Ticket expiration (now < expires_at)   -> OK
    3. Ticket provider_eid matches own EID    -> OK
    4. Consumer signature (consumer_vk from
       ticket)                                -> OK
    5. Select first mutually supported suite:
       hybrid                                 -> OK
```

Message 2: SuiteSelect (P -> C)

```
+-----------------------------------------------+
| selected_suite:                               |
|   "CIRP_X25519MLKEM768_ED25519_CHACHA20P..."|
| provider_signature: Ed25519(provider_sk,      |
|   SHA-256(selected_suite))                    |
+-----------------------------------------------+
```

```
  Consumer validates:
    1. Provider signature                     -> OK
    2. Selected suite in original offer       -> OK
       (downgrade protection)
```

Message 3: KeyExchange (C -> P)

```
+-----------------------------------------------+
| magic: "AIKX" (0x41494B58)                    |
```

```
   | session_id (16 bytes)                          |
   | role: 0x01 (Consumer)                          |
   | x25519_ephemeral_pk (32 bytes)                 |
   | mlkem768_encapsulation_key (1184 bytes)        |
   | signature: Ed25519(consumer_sk,                |
   |   session_id || role || x25519_pk || mlkem)    |
   +------------------------------------------------+

   Message 4: KeyExchange (P -> C)
   +------------------------------------------------+
   | magic: "AIKX" (0x41494B58)                     |
   | session_id (16 bytes)                          |
   | role: 0x02 (Provider)                          |
   | x25519_ephemeral_pk (32 bytes)                 |
   | mlkem768_ciphertext (1088 bytes)               |
   | signature: Ed25519(provider_sk,                |
   |   session_id || role || x25519_pk || ct)       |
   +------------------------------------------------+

   Key Derivation (both parties):
     classical_ss = X25519(my_sk, peer_pk)       // 32 bytes
     pq_ss = ML-KEM-768-Decaps(ct, my_dk)        // 32 bytes
             or ML-KEM-768-Encaps output

     ikm = classical_ss || pq_ss                 // 64 bytes
     PRK = HKDF-Extract(
       salt = session_id,                         // 16 bytes
       ikm  = ikm)                                // 64 bytes
     session_key = HKDF-Expand(
       PRK,
       info = "cirp-hybrid-kx"
              || "CIRP_X25519MLKEM768_ED25519_CHACHA20POLY1305_SHA256"
              || consumer_eid
              || provider_eid,
       L = 32)                                    // 256-bit key

   Message 5: First Encrypted Frame (C -> P)
   +------------------------------------------------+
   | magic: "AICF" (0x41494346)                     |
   | session_id (16 bytes)                          |
   | counter: 0x0000000000000001 (8 bytes, BE)      |
   | nonce (12 bytes, derived from counter)         |
   | ciphertext (N bytes, ChaCha20-Poly1305         |
   |   encrypted invocation envelope)               |
   | tag (16 bytes, Poly1305 auth tag)              |
   +------------------------------------------------+
```

After Message 5, the encrypted session is fully operational. All subsequent invocation and fulfillment envelopes are carried in AICF frames with incrementing counters.

Author’s Address

Saurabh Verma
Email: saurabh.sbay@gmail.com