Network Working Group
Internet-Draft
Intended status: Experimental
Expires: July 21, 2026

Peavey Koding
Independent Researcher

January 21, 2026

Kaspa Kinesis Transport Protocol (KKTP)
draft-koding-kktp-00

Status of This Memo

Abstract

The Kaspa Kinesis Transport Protocol (KKTP) is a serverless, CGNAT-compatible secure messaging protocol that uses the Kaspa BlockDAG as a decentralized mailbox for peer discovery, key agreement, and encrypted message relay. KKTP enables secure, replay-protected, and publicly verifiable communication between peers without STUN/TURN, centralized servers, or direct IP connectivity. This document specifies the protocol, data structures, cryptographic primitives, security properties, and operational considerations for KKTP.

Table of Contents

1. Introduction

Kaspa Kinesis Transport Protocol (KKTP) is a Kaspa-native, serverless, CGNAT-compatible secure messaging protocol. "Serverless" in the networking sense: KKTP requires no centralized servers, relays, or STUN/TURN infrastructure. It relies only on the Kaspa network, whose economic incentives (fees, block inclusion) are inherent to all blockchain-based systems.

It uses the Kaspa BlockDAG as a decentralized mailbox for:

- Peer discovery
- Key agreement
- Encrypted message relay

KKTP enables CGNAT-to-CGNAT communication without STUN/TURN, hole-punching, or centralized servers by leveraging the BlockDAG as an immutable, globally accessible buffer, using:

- Diffie-Hellman (DH) for key agreement
- VRF binding for anti-MITM
- AEAD for confidentiality and integrity

KKTP is application-agnostic: it can be used for games, chat, coordination, or any protocol that needs secure, ordered, replayable messaging.

Threat Model:
The adversary model, security assumptions, explicit non-goals, and attack mitigations for KKTP are defined in the companion document: "KKTP Threat Model".

2. Goals

- Serverless Connectivity:

Establish secure channels behind restricted NATs using only Kaspa.

- Public Verifiability:
  All discovery and session setup is publicly auditable on-chain.
  Public verifiability applies to session establishment and message
  attribution, not to encrypted message contents.

- Confidentiality & Integrity:
  End-to-end encryption using modern AEAD primitives.

- Replay Protection & Ordering:
  Strict sequence enforcement for BlockDAG environments.

- Anti-MITM:
  VRF binding prevents key-substitution attacks (see KKTP Threat Model,
  Section "Active Adversaries and MITM").

3. Entities

- Initiator:
  Publishes a discovery broadcast and accepts connections.

- Responder:
  Responds to a discovery broadcast to establish a session.

- Observer:
  Any third party verifying session setup or message integrity.

- Kaspa Network:
  Provides the public bulletin board (BlockDAG) for anchors and mailbox
  messages.

- Identity Mapping:
  In all direction fields and AAD calculations, the Initiator is A and
  the Responder is B.

4. Cryptographic Primitives

The following primitives are protocol constants:

- DH Key Exchange:
  X25519 (or secp256k1 ECDH if explicitly agreed per deployment).

- Signing Scheme:
  Ed25519 (or ECDSA over secp256k1 if explicitly agreed per deployment).

- Hash Function:
  BLAKE2b-256 (32-byte output).

- AEAD:
  XChaCha20-Poly1305.

- Nonces:
  MUST be 192-bit values generated via a CSPRNG. Nonce reuse with the
  same session key is strictly forbidden.

- KDF:
  HKDF-BLAKE2b.

- VRF:
  Deployment-specific (e.g., Kaspa+Bitcoin+QRNG folding).
  Deployments defining a VRF MUST specify a construction that provides
  public verifiability and collision resistance. The security goals and
  limitations of VRF binding are defined in the KKTP Threat Model.

Note: In KKTP, the term ’VRF binding’ refers to a publicly verifiable

binding of protocol keys to immutable anchors; it does not require unpredictability beyond collision resistance unless specified by the deployment.

All implementations of a given deployment MUST use the same concrete algorithms.

5. Data Structures

5.1 Canonical Encoding

All payload fields that are hashed or signed MUST be encoded in a canonical JSON format following RFC 8785 (JCS), except where explicitly excluded.

Implementations MUST ensure:
- UTF-8 encoding
- Object keys sorted lexicographically by key name
- No extra fields beyond those specified in the schema, except under the ‘meta‘ object
- No pretty-printing: no extraneous whitespace, newlines, or indentation
- Numbers encoded in decimal, no leading zeros, no scientific notation
- Booleans encoded as ‘true‘ / ‘false‘
- ‘null‘ only where explicitly allowed

All hex fields MUST be lowercase, even-length, with no ‘0x‘ prefix. Unless otherwise stated, the hash/signature input is the raw UTF-8 bytes of the canonical JSON encoding of the object with the signature field omitted AND with the ‘meta‘ object excluded.

5.2 Discovery Anchor

SID Entropy: The sid MUST be a high-entropy value (e.g., UUID v4 or 256-bit random hex string) to avoid collisions across different sessions between the same two peers. The sid also serves as a session-unique, non-secret HKDF salt.

JSON
```
{
  "type": "discovery",
  "version": 1,
  "sid": "<unique_session_id>",
  "pub_sig": "<hex>",
  "pub_dh": "<hex>",
  "vrf_value": "<hex|null>",
  "vrf_proof": "<hex|null>",
  "meta": {
    "game": "string",
    "version": "string",
    "expected_uptime_seconds": 3600
  },
  "sig": "<hex>"
}
```

Note: The ‘sig_resp‘ field is computed over the canonical JSON encoding of the object with the ‘sig_resp‘ field omitted. As this object contains no ‘meta‘ field, no additional exclusions apply.

5.3 Response Anchor

JSON
```
{
  "type": "response",
  "version": 1,
  "sid": "<unique_session_id>",
  "initiator_pub_sig": "<hex>",
```

```
  "initiator_pub_dh": "<hex>",
  "pub_sig_resp": "<hex>",
  "pub_dh_resp": "<hex>",
  "vrf_value": "<hex|null>",
  "vrf_proof": "<hex|null>",
  "sig_resp": "<hex>"
}
```

Note: The sig_resp field is computed over the canonical JSON encoding of the object with the sig_resp field omitted.

5.4 Mailbox Message (On-Chain Encrypted Packet)

JSON

```
{
  "type": "msg",
  "version": 1,
  "sid": "<unique_session_id>",
  "mailbox_id": "<hex>",
  "direction": "AtoB|BtoA",
  "seq": 0,
  "nonce": "<hex>",
  "ciphertext": "<hex>"
}
```

5.5 Session End Anchor

Published by either party. A session_end anchor MUST be signed using the pub_sig or pub_sig_resp associated with the session tuple.

JSON

```
{
  "type": "session_end",
  "version": 1,
  "sid": "<unique_session_id>",
  "pub_sig": "<hex>",
  "reason": "string",
  "sig": "<hex>"
}
```

Note: The sig field is computed over the canonical JSON encoding of the object with the sig field omitted.

6. Protocol Flow

6.1 Discovery & Handshake

Initiator:

- Generate long-term signing keypair (priv_sig, pub_sig) (or reuse existing).
- Generate ephemeral DH keypair (priv_dh, pub_dh).
- Optionally compute VRF binding:
  vrf_input = H(pub_sig || pub_dh || sid);
  vrf_value, vrf_proof = VRF(vrf_input).
  VRF binding mitigates post-publication key substitution under the adversary assumptions defined in the KKTP Threat Model.
- Construct Discovery Anchor (without sig), canonicalize, and sign:
  sig = Sign(priv_sig, canonical_json_without_sig).
- Publish as Kaspa transaction payload.
- Monitor Kaspa DAG for Response Anchors matching sid.

Responder:

- Scan Kaspa for Discovery Anchors matching criteria (e.g., meta.game, meta.version, meta.expected_uptime_seconds).

- For the chosen discovery:
  - Verify sig using pub_sig; optionally verify vrf_value / vrf_proof against vrf_input = H(pub_sig || pub_dh || sid).
  - Generate signing keypair (priv_sig_resp, pub_sig_resp) (or reuse existing).
  - Generate ephemeral DH keypair (priv_dh_resp, pub_dh_resp).
  - Optionally compute VRF binding: vrf_input = H(initiator_pub_sig || initiator_pub_dh || pub_sig_resp || pub_dh_resp || sid); vrf_value, vrf_proof = VRF(vrf_input).
  - Construct Response Anchor (without sig_resp), canonicalize, and sign: sig_resp = Sign(priv_sig_resp, canonical_json_without_sig_resp).
  - Publish as Kaspa transaction payload.

6.2 Session Key Derivation (HKDF)

Once both anchors are visible:

- Initiator computes K = DH(priv_dh, pub_dh_resp).
- Responder computes K = DH(priv_dh_resp, pub_dh).
- Both values are identical due to DH symmetry.
- Salt: Use the sid as the HKDF salt.
- Info: Concatenate pub_sig and pub_sig_resp.

  The Initiator’s pub_sig MUST appear first in all concatenations involving (pub_sig, pub_sig_resp), regardless of message direction or which peer performs the derivation.

- Derive: K_session = HKDF-Expand(HKDF-Extract(salt, K), info, 32).

6.3 Mailbox ID

The unique filter for on-chain packets is derived as:

mailbox_id = H(pub_sig || pub_sig_resp || sid)

The Initiator’s pub_sig MUST appear first in all concatenations involving (pub_sig, pub_sig_resp), including mailbox_id derivation.

6.4 Embedding in Kaspa Transactions

All KKTP messages are embedded in the payload field of Kaspa transactions.

Prefix:
All payloads MUST begin with the ASCII prefix "KKTP:".

Anchors:
"KKTP:ANCHOR:" || <canonical JSON>.

Messages:
"KKTP:" || mailbox_id_hex || ":" || <canonical JSON>.

Payload Limits:
The total payload size MUST NOT exceed Kaspa’s payload limit (~32 KB). If application-level data exceeds this limit, it MUST be chunked at the application layer into multiple msg packets with increasing seq.

6.5 Mailbox Detection and Scanning

Peers MUST iterate over each new block:

- If the payload does not start with "KKTP:", ignore it.

- If "KKTP:ANCHOR:", parse and process as an Anchor.

- If "KKTP:[hex]:", compare hex to local mailbox_id. If it matches, verify the AEAD tag, decode JSON, verify type == "msg", and process.

6.6 Encrypted Messaging

For each direction, peers maintain a local seq starting at 0.

Increment seq:
Each new message MUST use a strictly increasing sequence number.

Nonce:
192-bit CSPRNG nonce.

AAD Binding:
Associated Data MUST be included in every AEAD call to prevent reflection:
AAD = mailbox_id || direction || seq

Encrypt:
Produce ciphertext using XChaCha20-Poly1305 with K_session, nonce, and AAD.

Construct:
Build the Mailbox Message JSON as in Section 5.4.

Embed:
"KKTP:" || mailbox_id_hex || ":" || <canonical JSON>.

Decryption Hardening:
Implementations MUST verify the AEAD authentication tag before attempting to parse or allocate significant memory for the decrypted payload.

AAD Encoding:
The AAD MUST be constructed as the concatenation of:
- mailbox_id as raw hash bytes,
- direction encoded as the UTF-8 string "AtoB" or "BtoA",
- seq encoded as an unsigned 64-bit big-endian integer.

6.7 Reference Pseudocode (Informative)

```
# ---------------------------------------------------------------
# Discovery (Initiator)
# ---------------------------------------------------------------

function publish_discovery():
  priv_sig, pub_sig = gen_signing_keypair()
  priv_dh,  pub_dh  = gen_dh_keypair()
  sid = random_session_id()

  vrf_value, vrf_proof = optional_vrf(pub_sig, pub_dh, sid)

  obj = {
      type: "discovery",
      version: 1,
      sid: sid,
      pub_sig: pub_sig,
      pub_dh: pub_dh,
      vrf_value: vrf_value,
      vrf_proof: vrf_proof,
      meta: {...}
  }

  sig = sign(priv_sig, canonical(obj without sig))
```

```
  obj.sig = sig

  publish_to_kaspa("KKTP:ANCHOR:" || canonical(obj))


# ---------------------------------------------------------------
# Response (Responder)
# ---------------------------------------------------------------

function handle_discovery(anchor):
  if not verify_signature(anchor.pub_sig, anchor):
      return reject

  if not optional_verify_vrf(anchor):
      return reject

  priv_sig_resp, pub_sig_resp = gen_signing_keypair()
  priv_dh_resp,  pub_dh_resp  = gen_dh_keypair()

  vrf_value, vrf_proof = optional_vrf(
      anchor.pub_sig,
      anchor.pub_dh,
      pub_sig_resp,
      pub_dh_resp,
      anchor.sid
  )

  resp = {
      type: "response",
      version: 1,
      sid: anchor.sid,
      initiator_pub_sig: anchor.pub_sig,
      initiator_pub_dh: anchor.pub_dh,
      pub_sig_resp: pub_sig_resp,
      pub_dh_resp: pub_dh_resp,
      vrf_value: vrf_value,
      vrf_proof: vrf_proof
  }

  sig_resp = sign(priv_sig_resp, canonical(resp without sig_resp))
  resp.sig_resp = sig_resp

  publish_to_kaspa("KKTP:ANCHOR:" || canonical(resp))


# ---------------------------------------------------------------
# Session Key Derivation (Both Peers)
# ---------------------------------------------------------------

function derive_session_keys(anchorA, anchorB):
  if I_am_initiator:
      K = DH(priv_dh, anchorB.pub_dh_resp)
  else:
      K = DH(priv_dh_resp, anchorA.pub_dh)

  salt = anchorA.sid
  info = anchorA.pub_sig || anchorB.pub_sig_resp   # A first

  K_session = HKDF(salt, K, info)
  mailbox_id = H(anchorA.pub_sig || anchorB.pub_sig_resp || salt)

  return K_session, mailbox_id


# ---------------------------------------------------------------
# Sending a Message
```

```
# ---------------------------------------------------------------

function send_msg(direction, plaintext):
  seq[direction] += 1
  nonce = random_192bit()

  AAD = mailbox_id || direction || encode_u64(seq[direction])

  ciphertext = AEAD_Encrypt(K_session, nonce, AAD, plaintext)

  msg = {
      type: "msg",
      version: 1,
      sid: sid,
      mailbox_id: mailbox_id,
      direction: direction,
      seq: seq[direction],
      nonce: nonce,
      ciphertext: ciphertext
  }

  publish_to_kaspa("KKTP:" || mailbox_id_hex || ":" || canonical(msg))


# ---------------------------------------------------------------
# Receiving a Message
# ---------------------------------------------------------------

function handle_msg(msg):
  if msg.mailbox_id != mailbox_id:
      return ignore

  if msg.seq <= expected_seq[msg.direction]:
      return reject_replay

  AAD = mailbox_id || msg.direction || encode_u64(msg.seq)

  if not AEAD_Verify(K_session, msg.nonce, AAD, msg.ciphertext):
      return reject_invalid

  buffer[msg.direction][msg.seq] = msg

  while buffer contains expected_seq[msg.direction]:
      deliver(buffer[msg.direction][expected_seq])
      delete buffer entry
      expected_seq[msg.direction] += 1


# ---------------------------------------------------------------
# Session End
# ---------------------------------------------------------------

function handle_session_end(anchor):
  if not verify_signature(anchor.pub_sig, anchor):
      return reject

  stop_sending()
  stop_receiving()
  zeroize(priv_dh)
  zeroize(K_session)
```

6.8 Session State Machine (Informative)

Each KKTP session progresses through the following states:

INIT:

No anchors observed. No keys derived. No mailbox_id defined.

DISCOVERED:
A Discovery Anchor has been published or received. pub_sig, pub_dh, and sid are known. No session keys exist yet.

HANDSHAKE:
Both Discovery and Response Anchors are visible. All signatures and (optional) VRF proofs have been verified. K_session and mailbox_id are derived.

ACTIVE:
Encrypted messaging is permitted. seq counters advance independently for each direction. Replay protection, ordering, and AEAD verification rules apply.

FAULTED:
A protocol violation has occurred (e.g., missing seq gap timeout, buffer overflow, invalid AEAD tag, or inconsistent anchors). Implementations MAY terminate the session or request retransmission if supported.

CLOSED:
A session_end anchor has been observed. Implementations MUST stop sending or accepting new messages and securely erase priv_dh and K_session.

State Transitions:

INIT -> DISCOVERED
Upon observing or publishing a valid Discovery Anchor.

DISCOVERED -> HANDSHAKE
Upon observing a valid Response Anchor with matching sid.

HANDSHAKE -> ACTIVE
After deriving K_session and mailbox_id.

ACTIVE -> FAULTED
Upon replay, invalid AEAD tag, buffer overflow, or gap timeout.

ACTIVE -> CLOSED
Upon observing a valid session_end anchor.

FAULTED -> CLOSED
Upon local termination or receipt of session_end.

Invariant Mapping:

I1 (shared K_session) holds in HANDSHAKE and ACTIVE.
I2 (shared mailbox_id) holds in HANDSHAKE and ACTIVE.
I3 (consistent message ordering) applies only in ACTIVE.
I4 (public verifiability) applies from DISCOVERED through CLOSED.

7. Security & Correctness Rules

7.1 Replay Protection

Implementations MUST reject any message with a duplicate seq or a reused nonce per direction. Maintain the highest seen seq per direction and discard older or duplicate messages. Replay and reordering attack considerations are detailed in the KKTP Threat Model.

7.2 Ordering and DAG Reassembly

Kaspa is a BlockDAG; seq = n+1 may arrive before seq = n.

The Buffer:
Maintain a reassembly buffer for each direction.

Out-of-Order Handling:
Store out-of-order messages in the buffer and wait.

Strict Processing:
Application logic MUST only process messages in contiguous, strictly increasing order.

Decryption Failure:
If a message fails AEAD decryption, implementations MUST discard it and MUST NOT increment the expected seq.

Faults:
If a gap persists beyond an application-defined timeout (e.g., 30-60 seconds or 100 blocks), the session is "Faulted." Implementations MAY request a re-transmit (if supported) or terminate the session.

Buffer Limits:
Implementations MUST cap the out-of-order buffer to N messages or M bytes per direction.

7.3 Anti-MITM via VRF Binding

If VRF binding is used, implementations MUST verify that all signing and DH public keys present in both anchors match the VRF input.

In deployments where the VRF incorporates externally anchored entropy (e.g., QRNG, Bitcoin block hashes, and Kaspa block hashes folded together), VRF binding provides strong protection against post-publication key substitution, replay, and backdating attacks, and makes significant delay or reordering detectable to observers.

VRF binding does not replace identity authentication; full resistance to active man-in-the-middle attacks still requires trusted public keys or out-of-band verification.

7.4 Signature Verification

All anchors MUST have valid signatures:
-Discovery Anchor signed by priv_sig.
-Response Anchor signed by priv_sig_resp.
-Session End Anchor signed by the corresponding priv_sig.
-Implementations MUST reject anchors with invalid signatures.

7.5 Confidentiality & Integrity

All mailbox messages MUST be encrypted with XChaCha20-Poly1305 using K_session. Decryption failures MUST be treated as protocol violations and the message discarded.

7.6 Session Uniqueness

The tuple (sid, pub_sig, pub_sig_resp) uniquely identifies a session. Reuse of the same tuple with different DH keys MUST be treated as a new session.

7.7 Session Termination & Forward Secrecy

A session_end anchor signals that the mailbox is closed.

After observing and validating a session_end anchor, implementations MUST:

- stop sending and accepting new messages (after a grace period if defined),
- treat the mailbox as closed,
- securely erase (zero out) priv_dh and K_session from memory.

The meta.expected_uptime_seconds field provides a hint to peers but is not enforced by the protocol.

7.8 Key Separation

Implementations SHOULD NOT use the same keypair for both signing (Ed25519) and Diffie-Hellman (X25519). Distinct ephemeral DH keys provide Forward Secrecy; reusing long-term keys risks cross-protocol vulnerabilities.

KKTP intentionally does not provide cryptographic deniability; see the KKTP Threat Model and Section 10.6.

7.9 Strict Canonicalization

Any deviation from the lexicographical JSON sorting rules defined in Section 5.1 MUST result in signature verification failure.

Implementations MUST use a strict JSON serializer that does not add extra whitespace, indentation, or re-order fields.

7.10 Pruning and Archiving

Kaspa nodes prune historical data. If a session requires long-term state, implementations MUST archive messages locally because historical mailbox data will eventually be purged from the network.

8. Invariants

I1: For a given session, both peers derive the same K_session or fail entirely.

I2: All valid messages for a session share the same mailbox_id.

I3: No two honest peers will accept different sequences of messages for the same session.

I4: Any observer can verify that a session was established between specific keys and that all observed messages are attributable to that session.

7.11 Edge Case Handling (Informative)

The following table summarizes required behavior under common edge conditions. These rules are derived from Sections 6 and 7 and are provided for implementer clarity.

Condition	Required Behavior
Forked blocks in the DAG	Re-scan tips; process messages only once per unique block hash. Maintain replay protection via seq and nonce.
Maximum message size exceeded	Reject the message. Applications MUST chunk data at the application layer before embedding in KKTP packets.
Out-of-order buffer overflow	Treat session as "Faulted" per Section 7.2. Implementations MAY request retransmission or terminate.

Failed VRF verification	Reject the anchor. Do not derive session keys.
Failed signature verification	Reject the anchor or message. Do not process or store invalid data.
AEAD authentication failure	Discard the message. MUST NOT advance seq or allocate memory for plaintext.
Duplicate seq or reused nonce	Reject as replay. MUST NOT decrypt or deliver to application.
Missing seq (gap persists)	After timeout, mark session "Faulted" and MAY terminate.

9. Extensibility

Additional fields (e.g., capabilities, QoS hints, application metadata) MAY be added under the ‘meta‘ object.

The ‘meta‘ object MUST NOT affect hashing, signatures, key derivation, mailbox_id derivation, or any security-critical computation.

Any extension that affects protocol semantics MUST be versioned via the top-level ‘version‘ field.

Multiple mailboxes (e.g., for different channels) MAY be derived by including a channel ID in the mailbox_id hash:

```
mailbox_id_channel = H(mailbox_id || channel_id)
```

10. Security Considerations

This section summarizes key risks and mitigations. A complete adversary model, including explicit non-goals and attack analyses, is defined in the companion document "KKTP Threat Model".

10.1 Nonce Misuse

Warning: Encrypting two different messages with the same K_session and nonce leads to a total loss of confidentiality.

Mitigation: Always use CSPRNG for 192-bit nonces. Never use a simple counter unless combined with a unique per-session salt.

10.2 Forward Secrecy

KKTP achieves Forward Secrecy by using ephemeral DH keys. Once a session ends, priv_dh and K_session MUST be securely erased from memory. Storing on disk increases compromise risks.

10.3 Metadata Leakage

Message content is encrypted, but mailbox_id, sid, and public keys are visible on-chain. Observers can see communication patterns. For privacy, use burnable ephemeral identity keys per session.

10.4 Canonical JSON Mismatches

Common failure cause: Mismatches in canonicalization.

Prevention: Use a library guaranteeing RFC 8785 or equivalent sorting. Even minor deviations cause signature failures.

10.5 DoS via Mailbox Flooding

KKTP relies on Kaspa’s transaction fees to make flooding expensive. Implementations MUST perform the Poly1305 integrity check before any heavy processing; discard if tag invalid.

10.6 Lack of Cryptographic Deniability

KKTP does not provide cryptographic deniability. Session establishment is publicly anchored and signed, allowing third parties to verify that specific public keys participated in a given session. This property is intentional and enables public verifiability and auditability, but it means participants cannot plausibly deny session participation after the fact.

11. Quick Start: 6 Steps

1. Keys:
   Generate Ed25519 (identity) and X25519 (session) keypairs.

2. Discovery:
   Post a Discovery Anchor with a high-entropy sid.

3. Handshake:
   Monitor Kaspa for a Response Anchor matching your sid.

4. Secret:
   Compute the DH shared secret and derive K_session via HKDF.

5. Communicate:
   Send and receive msg packets using the mailbox_id and strict AAD binding.

6. Terminate:
   Post a session_end anchor and wipe session keys from memory when finished.

11.1 Session Lifecycle Diagram (Informative)

```
A = Initiator        K = Kaspa DAG        B = Responder
------------------------------------------------------------

A: gen keys, VRF?
A -> K: [Discovery Anchor]

                                    B: scan, verify
                                    B: gen keys, VRF?
K -> B: [Discovery Anchor]

B -> K: [Response Anchor]

A: derive K_session
B: derive K_session
A,B: mailbox_id = H(pub_sig || pub_sig_resp || sid)

------------------------------------------------------------
Encrypted Messaging
------------------------------------------------------------

A -> K: [msg A->B, seq, nonce, AEAD]
                                    B: verify, decrypt, deliver

B -> K: [msg B->A, seq, nonce, AEAD]
                                    A: verify, decrypt, deliver
```

```
------------------------------------------------------------
Session End
------------------------------------------------------------

A or B -> K: [session_end]

A,B: stop sending, close mailbox,
     erase priv_dh and K_session
```

12. Licensed under the MIT License.

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.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, May 2017.

[RFC8785] Jones, M., "JSON Canonicalization Scheme (JCS)", RFC 8785, June 2020.

[HKDF] Krawczyk, H., and Eronen, P., "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, May 2010.

13.2 Informative References

[X25519] Langley, A., Hamburg, M., and Turner, S., "Elliptic Curves for Security", RFC 7748, January 2016.

[XCHACHA] Aumasson, J-P., et al., "XChaCha20-Poly1305: AEAD Construction", IETF CFRG Internet-Draft (work in progress).

[VRF] Goldberg, S., et al., "Verifiable Random Functions", various academic sources.

Authors’ Addresses

Peavey Koding
Independent Researcher
Email: peavey2787@yahoo.com
GitHub: https://github.com/peavey2787