Independent Submission
Internet-Draft
Intended status: Informational
Expires: 3 November 2026

D. Samsonov
May 2026

UDPN: UDP Datagram Privacy Network Protocol Version 1.0
draft-udpn-protocol-00

Abstract

This document specifies the UDP Datagram Privacy Network (UDPN) protocol, version 1.0. UDPN provides an authenticated, encrypted Layer 3 tunnel over UDP with traffic obfuscation designed to resist deep packet inspection (DPI) and active probing. All packets are wrapped in DTLS 1.2 ApplicationData records. The protocol uses the Noise_NK handshake pattern with X25519 Diffie-Hellman and ChaCha20-Poly1305 AEAD encryption.

Table of Contents

1. Introduction

UDPN creates a Layer 3 IP tunnel over UDP suitable for networks where deep packet inspection, port blocking, or active probing is used against tunneling traffic.

Three threat models are addressed:

a. Passive observation: all payload bytes are encrypted and computationally indistinguishable from random data.

b. Active probing: the server silently discards every packet it cannot cryptographically verify. An adversary receives no response whatsoever, making the endpoint indistinguishable from a closed UDP port.

c. Traffic correlation: periodic port hopping changes the UDP 5-tuple; random per-packet padding obscures payload lengths; jittered keepalive intervals resist timing fingerprinting.

All packets are formatted as DTLS 1.2 ApplicationData records to blend with legitimate DTLS/CoAP traffic on the wire.

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD 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.

Initiator The client endpoint that initiates the session.

Responder The server endpoint that accepts sessions.

Session A cryptographic context active between a successful Noise handshake and the next teardown event.

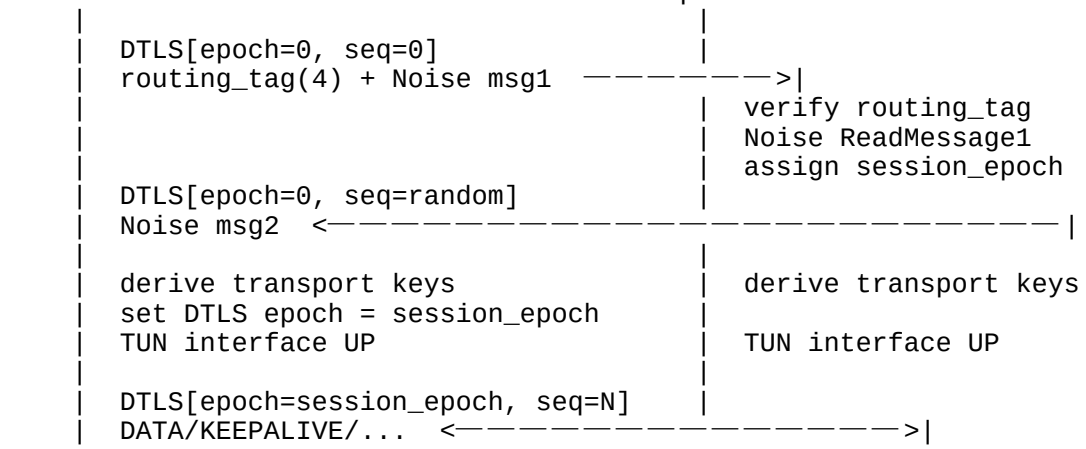
DTLS Epoch A 16-bit session identifier assigned by the Responder. Stable for the whole session lifetime. Not related to DTLS rekeying.

Hop Epoch A 16-bit counter incremented by the Initiator at each port-hop event, carried inside the encrypted payload.

Nonce The 64-bit value used as the ChaCha20-Poly1305 nonce, carried in the DTLS sequence-number field.

3. Protocol Overview

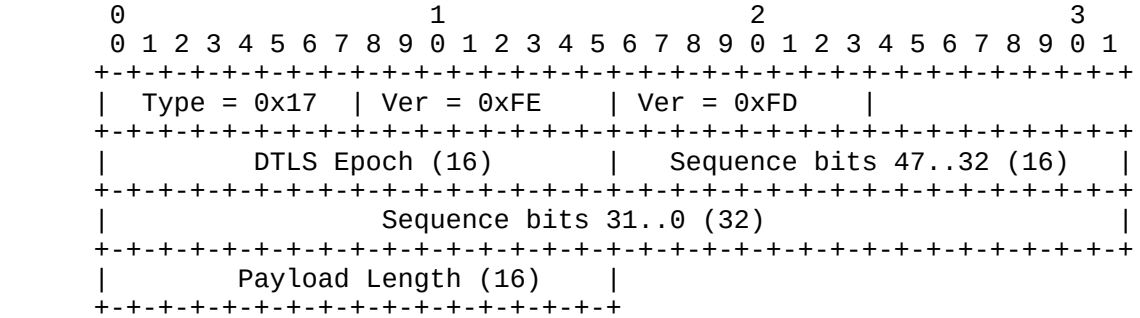
A UDPN session proceeds as follows:



4. Packet Format

4.1. DTLS Record Header

Every UDPN packet starts with a 13-byte DTLS 1.2 record header:



Type (8 bits) MUST be 0x17 (DTLS ApplicationData). Packets with any other value MUST be silently discarded.

Version (16 bits) MUST be 0xFE 0xFD (DTLS 1.2) per [RFC6347]. Mismatch causes silent discard.

DTLS Epoch (16 bits) 0x0000 for handshake packets; session_epoch for transport packets. Stable for the session lifetime.

Sequence (48 bits, big-endian) Serves as both DTLS sequence number and ChaCha20-Poly1305 nonce (zero-extended to 64 bits, stored little-endian per RFC 8439). MUST be monotonically increasing per session. MUST NOT be reset on port hops.

Length (16 bits) Byte count of the following payload.

4.2. Handshake Packets

Handshake packets use DTLS Epoch = 0. A packet with DTLS Epoch = 0 and DTLS payload length less than 4 bytes MUST be silently discarded. Such a packet cannot contain a valid routing_tag (msg1 minimum: 36+ bytes) nor a complete Noise msg2 (minimum 32 bytes for e_pub alone).

The msg1 payload layout is:

Offset	Size	Field	
0	4	routing_tag	BLAKE2s(e_pub||s_pub)[0:4]
4	32	e_pub	Initiator ephemeral public key
36	*	ciphertext	ChaCha20-Poly1305(inner_payload)
36+*	16	AEAD_tag	Poly1305 authentication tag

The msg2 payload layout (no routing_tag) is:

Offset	Size	Field	
0	32	e_pub	Responder ephemeral public key
32	*	ciphertext	ChaCha20-Poly1305(inner_payload)
32+*	16	AEAD_tag	Poly1305 authentication tag

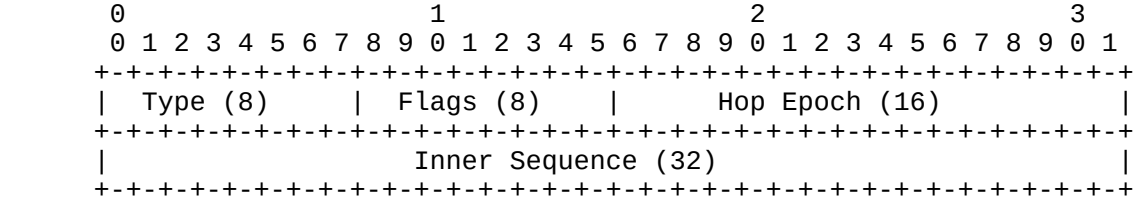
In both messages the AEAD tag immediately follows the ciphertext, matching standard AEAD.Seal() output per [RFC8439].

4.3. Transport Packets

Transport packets use DTLS Epoch = session_epoch. Their payload is the output of a single AEAD.Seal() call: ciphertext followed immediately by the 16-byte Poly1305 authentication tag.

4.4. Inner Header

The first 8 bytes of every transport plaintext are the inner header:



Type (8 bits) 0x01 DATA, 0x06 KEEPALIVE, 0x07 KEEPALIVE_ACK, 0x08 DISCONNECT. Unknown values MUST be silently discarded.

Flags (8 bits) Reserved. MUST be 0x00 on send; MUST be ignored on receive.

Hop Epoch (16 bits, big-endian) Current port-hop counter, starting at 0. Wraps modulo 65536.

Inner Sequence (32 bits, big-endian) Per-direction monotonic counter for sliding-window replay protection. Starts at 0 after each new

session.

Total per-packet overhead: 13 (DTLS) + 8 (inner header) + 16 (AEAD tag) = 37 bytes, plus padding.

5. Cryptographic Design

5.1. Keypair Generation

Each connection uses a unique X25519 keypair generated by the Responder. The private key is kept exclusively by the Responder. The public key is distributed to the Initiator out-of-band (e.g., a configuration file). Key clamping follows [RFC7748] Section 5.

5.2. Noise_NK Handshake

UDPN uses the Noise Protocol Framework [NOISE] with:

```
Pattern:  NK
DH:       25519 (X25519, RFC 7748)
Cipher:   ChaChaPoly (ChaCha20-Poly1305, RFC 8439)
Hash:     SHA256

Full protocol name: Noise_NK_25519_ChaChaPoly_SHA256

Pattern NK:
   <- s           (premessage: Responder static public key)
   ...
   -> e, es       (msg1: Initiator ephemeral + DH)
   <- e, ee       (msg2: Responder ephemeral + DH)
```

The NK pattern provides: Responder authentication (Initiator knows its public key); Initiator anonymity (Responder learns nothing about Initiator identity); and forward secrecy (both ephemeral keys are discarded after the handshake).

Prologue: empty byte sequence (zero bytes).

5.3. Transport Keys

Upon completing the handshake, both parties call split() to derive two ChaCha20-Poly1305 keys (c1, c2). The Initiator sends with c1 and receives with c2. The Responder sends with c2 and receives with c1. The transport key is NOT rotated within a session; each new session derives fresh independent keys through new ephemeral DH.

5.4. AEAD Nonce Construction

ChaCha20-Poly1305 requires a 96-bit (12-byte) nonce, constructed as follows:

```
nonce[0..3]  = 0x00 0x00 0x00 0x00   (4 zero bytes)
nonce[4..11] = DTLS Sequence          (64-bit, little-endian)

Step-by-step:
(1) Read DTLS Sequence from wire: 6 bytes big-endian -> uint64
    (upper 16 bits = 0).
(2) Encode uint64 as little-endian into nonce[4..11].

Example: DTLS Sequence = 1 (0x000000000001 on wire)
   Decoded uint64:  0x0000000000000001
   nonce[4..11]:    01 00 00 00 00 00 00 00  (little-endian)
```

This follows [RFC8439] Section 2.4. Additional Data (AD) passed to AEAD is always empty.

5.5. Routing Tag

The Initiator prepends a 4-byte routing tag to msg1 to allow the Responder to identify the target connection without attempting O(N) Noise decryptions. The tag uses BLAKE2s-256 [BLAKE2]:

```
routing_tag = BLAKE2s-256(e_pub || s_pub)[0:4]
```

where e_pub is the Initiator's ephemeral public key and s_pub is the Responder's static public key for the target connection. The Responder scans connections computing BLAKE2s-256(e_pub || conn.s_pub)[0:4] until a match is found, then performs one full Noise decryption to authenticate.

Note: the routing tag does not eliminate the X25519 operation. It reduces Nx(X25519+AEAD) to NxBLAKE2s + 1x(X25519+AEAD). At N=1000 this is approximately 500ms reduced to 1ms for the lookup phase.

Collision probability is approximately N/2^32 per handshake attempt. On collision, both matching connections are tried; the AEAD step resolves the ambiguity.

6. Handshake

6.1. Initiator Handshake Message (msg1)

The Initiator sends a UDP packet with DTLS Epoch = 0 and Sequence = 0. The DTLS payload contains: routing_tag (4 bytes), followed by the Noise msg1 bytes (e_pub + ciphertext + tag).

The encrypted inner_payload contains:

Offset	Size	Field	
0	8	conn_id_hint	BLAKE2s derivative of conn id
8	2	hop_interval	Hop interval in minutes (uint16 BE)
10	4	pool_hash	HMAC-SHA256 of port pool (uint32 BE, first 4 bytes)
14	P	padding	P random bytes (P >= padding.min)

The conn_id_hint, hop_interval, and pool_hash fields are informational. The Responder validates minimum payload size (14 bytes) but does not use these values for session routing or authentication.

6.2. Responder Handshake Message (msg2)

The Responder replies with DTLS Epoch = 0 and a random Sequence value. The DTLS payload is the Noise msg2 bytes (e_pub + ciphertext + tag).

The encrypted inner_payload contains:

Offset	Size	Field	
0	2	session_epoch	DTLS epoch for this session (uint16 BE)
2	P	padding	P random bytes

session_epoch is a randomly chosen value in [0x0001..0xFFFE]. The Responder MUST NOT assign session_epoch = 0x0000, as zero is reserved to identify handshake packets; transport packets with epoch 0 would be indistinguishable from handshake packets. The Responder MUST also verify the epoch is not in use by another active session; if a collision is found, a new value MUST be drawn.

6.3. Post-Handshake State

After a successful exchange, both parties:

a. Derive transport keys via split().

b. Reset the DTLS Sequence (Noise nonce) counter to 0.

c. Reset the Inner Sequence counter to 0.

d. Reset the inner sliding-window replay state.

The Initiator additionally sets its DTLS Epoch to session_epoch and brings the TUN interface UP. The Responder records the session DTLS Epoch, Initiator src IP:port, and local UDP port, and brings its TUN interface UP. If a session already existed for this connection, it is evicted first.

7. Transport

7.1. Sending a Packet

1. Claim nonce N = dtlsSeqTx.fetch_add(1) atomically.

2. Claim Inner Sequence S = innerTxSeq.fetch_add(1) atomically.

3. Choose padding P = padding.min + PRNG(padding.max + 1).

4. Construct plaintext: inner_header (8 bytes) || L3 packet || P zero bytes.

5. Encrypt: ciphertext = AEAD_Encrypt(send_key, nonce=N, AD="", plaintext).

6. Construct DTLS record: Type=0x17, Ver=0xFEFD, Epoch=session_epoch, Seq=N, Length=len(ciphertext).

7. Enqueue for transmission.

7.2. Receiving a Packet

1. Check DTLS Type=0x17, Version=0xFEFD. On mismatch: DISCARD silently.

2. If DTLS Epoch = 0: route to handshake processing.

3. Look up active session by DTLS Epoch. If none: DISCARD silently.

4. ACL check (server only). On failure: DISCARD, increment acl_drop.

5. Decrypt: plaintext = AEAD_Decrypt(recv_key, nonce=DTLS_Seq, AD="", ciphertext). On failure: DISCARD, increment replay_drop.

6. Parse inner header from plaintext[0..7].

7. Sliding-window check on Inner Sequence. On failure: DISCARD, increment replay_drop.

8. Update keepalive watchdog (touch last-seen timestamp).

9. Server only: update routing state (see Section 9.3).

10. Dispatch on inner Type (see Section 7.3).

7.3. Inner Packet Types

DATA (0x01) Write payload bytes (offset 8+) to the TUN interface. The IP/IPv6 header length field defines the packet boundary; trailing padding is ignored.

KEEPALIVE (0x06) Respond asynchronously with KEEPALIVE_ACK.

KEEPALIVE_ACK (0x07) No action beyond the watchdog touch in step 8.

DISCONNECT (0x08) Stop keepalive manager, bring TUN DOWN, clear session state. Sender SHOULD transmit three copies to improve delivery probability.

8. Keepalive

Both endpoints send KEEPALIVE packets at random intervals drawn from [T*0.8, T*1.2] where T is the configured keepalive interval in seconds. Jitter prevents interval-based fingerprinting.

The watchdog fires after T * timeout_factor seconds of inactivity (default timeout_factor = 3, minimum 1). On timeout, the Initiator tears down the session and schedules reconnection; the Responder tears down and waits for a new handshake.

9. Port Hopping

9.1. Port Selection

Given a sorted pool P of N port numbers:

```
SelectPort(key, session_id, epoch, direction, P):
   mac = HMAC-SHA256(key=key,
                     data=uint64_BE(session_id) || uint16_BE(epoch)
                          || uint8(direction))
   index = uint16_BE(mac[0:2]) mod N
   return P[index]

direction = 0 for destination port
direction = 1 for source port
```

The direction byte ensures dst_port and src_port are derived independently, eliminating structural collisions for all pool sizes. If SelectPort(key, sid, epoch, 1, P) == SelectPort(key, sid, epoch, 0, P), the source port MUST be advanced to the next entry in the pool.

In UDPN v1.0, key = Responder static public key, session_id = 0.

9.2. Hop Procedure (Initiator)

At each hop_interval (minutes):

1. Increment hop_epoch by 1 (mod 65536).
2. Compute dst_port = SelectPort(s_pub, 0, hop_epoch, 0, pool).
3. Compute src_port = SelectPort(s_pub, 0, hop_epoch, 1, pool).
4. Update internal dstAddr to (server_ip, dst_port).
5. Rebind local UDP socket to src_port. On bind failure, try other pool ports (excluding dst_port). If all fail, skip this hop.
6. Begin sending with Hop Epoch = hop_epoch in the inner header.

The DTLS Sequence (Noise nonce) MUST NOT be reset on a hop event. Resetting would cause replay detection failures on the Responder.

9.3. Hop Reception (Responder)

In step 9 of Section 7.2, the Responder updates routing state as follows:

If inner.HopEpoch == current hop_epoch:

* Update dstAddr to packet's src IP:port.

* Update replyPort to packet's dst port.

If inner.HopEpoch != current hop_epoch, compute delta = (incoming - current) mod 65536:

delta in [1..4] — ACCEPT Update hop_epoch, dstAddr, replyPort. Touch keepalive watchdog.

delta in [5..32767] — REJECT hop_epoch, dstAddr, and replyPort are NOT updated. The packet payload IS delivered normally (AEAD already verified authenticity). Log a warning.

delta in [32768..65535] — ignore Old epoch (reordered packet). Process payload, ignore hop field.

After a hop, packets arriving with the old hop_epoch are still accepted as long as they pass AEAD verification (same session key across hops) and the inner sequence window check. No explicit grace timer is required.

10. Replay Protection

10.1. AEAD Layer

The DTLS Sequence is a monotonically increasing uint64 counter per session, never reset within a session. Replay of any packet with a previously used DTLS Sequence fails AEAD verification. This is the primary replay barrier.

10.2. Inner Sequence Window

A 1024-packet sliding window operates on the 32-bit Inner Sequence field, providing secondary duplicate detection for reordered packets (e.g., ECMP path changes or Wi-Fi retransmissions).

Let WINDOW_SIZE = 1024. Processing Inner Sequence S (uint32):

```
diff = (S - maxSeen) mod 2^32

if diff < 2^31:                        # S is newer than maxSeen
   if diff >= WINDOW_SIZE:
      bits = 0                          # far ahead -- reset bitmap
   elif diff > 0:
      bits <<= diff                     # slide window forward
   maxSeen = S
   bits[0] |= 1                         # mark S as received
   return ACCEPT

else:                                   # S is older than maxSeen
   offset = (maxSeen - S) mod 2^32
   if offset >= WINDOW_SIZE: return DISCARD   # too old
   if bit[offset] is set:   return DISCARD   # duplicate
   set bit[offset]
```

```
        return ACCEPT
```

The window is reset at each new session. Window size rationale: 1024 absorbs reordering from ECMP routing and Wi-Fi retransmission buffers without false positives. WireGuard uses 2048; OpenVPN uses 128. Implementation: the 1024-bit bitmap is stored as [16]uint64.

10.3. Handshake Replay

The Responder maintains a TTL cache of Initiator ephemeral public keys (e_pub) observed in msg1, with a TTL of 30 minutes. If an e_pub is seen a second time, the packet is silently discarded. Memory: approximately 32 bytes per entry; at 1 handshake/second over 30 minutes 57 KB.

11. Padding

Each outgoing packet includes random padding: pad_length = padding.min + PRNG(padding.max + 1), where PRNG is a non-cryptographic uniform PRNG (PCG algorithm). Padding length is not security-sensitive; only unpredictability of sizes is required. Default values: padding.min = 16, padding.max = 128. Padding bytes on the wire are zero; receivers ignore them.

12. Session Lifecycle

```
    IDLE ————————————————————————————————————————————————————————————> CONNECTING
    CONNECTING —(msg2 received)————————————————————————> ESTABLISHED
    CONNECTING —(timeout * 3 attempts)————————————> IDLE (reconnect delay)
    ESTABLISHED —(keepalive timeout)————————————————> IDLE (reconnect delay)
    ESTABLISHED —(DISCONNECT received)——————————————> IDLE (reconnect delay)
    ESTABLISHED —(hop timer fires)——————————————————> HOPPING -> ESTABLISHED
```

Initiator reconnect delay: configurable (default 5 seconds). On graceful shutdown (SIGTERM/SIGINT), three DISCONNECT packets are sent in quick succession, followed by a 300 ms drain delay before exit.

13. Security Considerations

13.1. Active Probing Resistance

The Responder MUST silently discard: packets with DTLS type other than 0x17; packets with DTLS version other than 0xFEFD; epoch-0 packets failing Noise msg1 decryption; transport packets failing AEAD verification; packets with epoch not matching any active session. Under no circumstances MUST the Responder send any response to an unauthenticated packet, including ICMP Port Unreachable.

13.2. Forward Secrecy

The Noise_NK handshake derives session keys from a combination of the Initiator’s fresh ephemeral key pair and the Responder’s fresh ephemeral key pair. Compromise of the Responder’s static private key after a session concludes does NOT reveal session keys.

13.3. Replay Attacks

Three independent mechanisms prevent replay: (a) handshake ephemeral key cache; (b) AEAD nonce monotonicity; (c) inner sequence sliding window.

13.4. Traffic Analysis

Mitigations: random per-packet padding obscures sizes; keepalive jitter (+/-20%) obscures timing; port hopping changes the UDP

5-tuple. Limitations: total traffic volume and packet rate are not obscured; an adversary observing traffic before and after a hop may correlate flows by timing proximity.

13.5. Choice of ChaCha20-Poly1305

ChaCha20-Poly1305 is preferred over AES-256-GCM because its software implementation is constant-time regardless of hardware AES support. AES-GCM without AES-NI instructions is vulnerable to cache-timing attacks [BERNSTEIN]. On virtualised platforms (KVM/QEMU), AES-NI passthrough cannot be universally guaranteed.

13.6. Path MTU Discovery

UDPN does not implement PMTUD for the outer UDP encapsulation; outer packets are sent without the DF bit. Inner oversized L3 packets are dropped and replaced with ICMP Fragmentation Needed (IPv4, [RFC1191]) or ICMPv6 Packet Too Big (IPv6, [RFC1981]) messages. Operators SHOULD configure tun_mtu = min(path_mtu, 1500) - 37.

14. Implementation Notes

14.1. TUN Interface

The TUN file descriptor MUST be opened without O_NONBLOCK. Reads are performed via blocking syscall with the goroutine locked to its OS thread (runtime.LockOSThread). TUN MTU = configured_mtu - 37.

14.2. Server Transport (epoll)

With N listening sockets (typically 257), a naive one-goroutine-per-socket model creates N OS threads blocked in recvfrom(2). The RECOMMENDED approach is edge-triggered epoll(7) with 2 worker goroutines, each draining ready file descriptors until EAGAIN.

14.3. Batch UDP Send (sendmmsg)

Implementations SHOULD use sendmmsg(2) to send multiple outgoing packets per syscall. A batch size of 32 is RECOMMENDED. Fall back to individual sendmsg(2) if unavailable.

14.4. Lock-Free Encrypt Path

For maximum throughput, the encrypt hot path SHOULD be lock-free:

* Transport state pointer: atomic.Pointer (nil = no session).

* DTLS epoch and Hop epoch: packed into a single atomic uint32.

* DTLS Sequence (Noise nonce): atomic uint64, incremented with Add.

* Inner Sequence: atomic uint32, incremented with Add.

Multiple goroutines can encrypt concurrently without mutex contention, each atomically claiming a unique nonce. The AEAD object (cipher.AEAD) is created once at session start and reused; Seal/Open are goroutine-safe.

15. IANA Considerations

This document has no IANA actions. UDPN uses UDP ports chosen by the operator. The ciphertext carried in DTLS content type 0x17 is indistinguishable from random data and is not registered with IANA.

16. References

16.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/rfc/rfc7748>.

[RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/rfc/rfc8439>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/rfc/rfc6347>.

16.2. Informative References

[NOISE] Perrin, T., "The Noise Protocol Framework, Revision 34", https://noiseprotocol.org/noise.html, 2018.

[BLAKE2] Aumasson, J-P., Neves, S., Wilcox-O’Hearn, Z., and C. Winnerlein, "BLAKE2: simpler, smaller, fast as MD5", https://www.blake2.net/, 2013.

[BERNSTEIN] Bernstein, D.J., "Cache-timing attacks on AES", https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[RFC1191] Mogul, J. and S. Deering, "Path MTU Discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <https://www.rfc-editor.org/rfc/rfc1191>.

[RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, DOI 10.17487/RFC1981, August 1996, <https://www.rfc-editor.org/rfc/rfc1981>.

Author’s Address

Denis Samsonov
Email: i@denjs.com
URI: https://denjs.com