LO Protocol - Specification for Formal Analysis

1. Scope

This document presents the Soliton cryptographic protocol at an abstract level for the purpose of formal security analysis. Concrete algorithm identifiers (X-Wing draft-09, ML-DSA-65, HKDF-SHA3-256, XChaCha20-Poly1305) are named where they constrain the security model but described only by their abstract properties. Implementation artifacts - serialization formats, language bindings, memory management - are omitted entirely.

The protocol addresses two-party end-to-end encrypted messaging and voice calls with hybrid classical/post-quantum security. It consists of five sub-protocols:

- **LO-Auth**: KEM-based key possession proof for server-side authentication.
- **LO-KEX**: Asynchronous KEM-based session establishment (analogous to X3DH/PQXDH).
- **LO-Ratchet**: Ongoing message encryption providing forward secrecy and break-in recovery, via a KEM-based adaptation of the Double Ratchet.
- **LO-Call**: Call encryption key derivation for E2EE voice/video calls, with ephemeral KEM for call-specific forward secrecy and an intra-call chain ratchet.
- **LO-Stream**: Streaming AEAD for bulk data (file transfer, media), providing chunked encryption with ordering, truncation resistance, and cross-stream isolation guarantees (§15, Theorem 13).

---

2. Cryptographic Primitives

2.1 Hybrid KEM (X-Wing)

X-Wing combines X25519 (classical) and ML-KEM-768 (post-quantum) via a combiner. Key pairs satisfy pk = (pk_M, pk_X), sk = (sk_M, sk_X). (Note: this tuple notation follows the draft-09 mathematical presentation order — ML-KEM-first. The LO wire encoding uses X25519-first: pk = pk_X ‖ pk_M, sk = sk_X ‖ sk_M, consistent with c = (ct_X ‖ c_M). The combiner input order ss_M ‖ ss_X likewise follows ML-KEM-first per draft-09 §5.3, independent of the LO wire layout.)

Encaps(pk) → (c, ss):

- Generate ephemeral X25519 key pair (ek_sk, ek_pk) ← X25519.KeyGen(); set ct_X := ek_pk. (The ephemeral public key ek_pk serves as the X25519 ciphertext component of c; draft-09 §5.3 names it ct_X, and that label is used hereafter.) Compute c_M, ss_M via ML-KEM.Encaps(pk_M); compute ss_X = DH(ek_sk, pk_X). (If the DH output is the identity element, it is included in the combiner as the all-zeros string without abortion.)
- c = (ct_X ‖ c_M); ss = H_3(ss_M ‖ ss_X ‖ ct_X ‖ pk_X ‖ Λ). (Note: the ML-KEM ciphertext c_M is not an input to the combiner; only the X25519 ephemeral public key ct_X appears, as specified in draft-09 §5.3.)
- Λ = 0x5c2e2f2f5e5c (ASCII: `\.//^\`, fixed 6-byte domain label, appended last)
- H_3 denotes SHA3-256.

Decaps(sk, c) → ss: parses c as (ct_X ‖ c_M), recomputes ss_M via ML-KEM.Decaps(sk_M, c_M) and ss_X = DH(sk_X, ct_X), then applies the identical combiner. The combiner requires pk_X (the holder's own X25519 public key), which is derived from sk_X and is not transmitted in c.

Implicit rejection: ML-KEM decapsulation uses implicit rejection (FIPS 203 §7.3) — a wrong ciphertext or wrong key produces a pseudorandom shared secret rather than an error. Decapsulation is infallible at the primitive level; a key/ciphertext mismatch propagates silently through KDF_Root and KDF_MsgKey until the AEAD layer detects it (`AEAD.Dec → ⊥` at §4.4 Step 8 or §5.4 Step 4). For Tamarin/ProVerif, `Decaps` cannot be modeled as returning ⊥ on failure — the detection point is the AEAD decryption, not the KEM.

For symbolic analysis (Tamarin, ProVerif), X-Wing may be treated as a single IND-CCA2 KEM, abstracting the combiner. For computational models (CryptoVerif), the combiner should be opened explicitly, with the security reduction proceeding from the hybrid security theorem under H_3 modeled as a random oracle.

2.2 Hybrid Signature Scheme

HybridSig combines Ed25519 (classical) and ML-DSA-65 (post-quantum). Key pairs satisfy pk = (pk_E, pk_P), sk = (sk_E, sk_P).

Sign(sk, m) → σ = (σ_E ‖ σ_P): both components computed independently and concatenated. ML-DSA-65 uses hedged signing via `sign_internal` (FIPS 204 §5.2 / Algorithm 2); fresh randomness is mixed per signing operation for fault-injection resistance. **FIPS 204 compatibility note**: The implementation calls `sign_internal` directly — the raw internal signing function with no context string or domain prefix. This is structurally incompatible with FIPS 204 §6.2 (`ML-DSA.Sign`, which prepends a context-dependent domain separator before calling `sign_internal`). A FIPS 204 §6.2 verifier expecting the domain-prefixed message format will reject Soliton ML-DSA-65 signatures. A formal model or test vector suite must use the `sign_internal` / `verify_internal` interface, not the §6.2 external interface. For adversary models that include fault injection, hedged signing provides resistance to differential fault analysis that deterministic signing does not. **RNG implication**: Every HybridSig.Sign invocation consumes randomness (from ML-DSA-65's hedged component). In the §8.2 RNG table, signing randomness is subsumed by the per-step time point: σ_SI at §4.3 and σ_SPK at bundle publication share the Corrupt(RNG) query of their respective protocol step. A Tamarin Sign rule should consume a Fr(~r) fact; however, since Corrupt(RNG, P, t) at a signing time point yields the signing randomness alongside all other randomness in that step, this adds no adversarial advantage beyond what existing corruption queries provide — the adversary with Corrupt(IK, P) can already sign arbitrary messages.

Verify(pk, m, σ) → 1 iff Ed25519.Verify(pk_E, m, σ_E) = 1 **and** ML-DSA.Verify(pk_P, m, σ_P) = 1. Both components are evaluated eagerly (no short-circuit), and the AND combination must be constant-time (e.g., bitwise AND on single-bit results) — a branching short-circuit leaks which component failed, enabling targeted forgery of only the weaker scheme. HybridSig is EUF-CMA if at least one component is EUF-CMA. For symbolic models, it may be treated as a single EUF-CMA signature scheme.

2.3 Key Derivation and MAC

BEn(x) denotes the big-endian encoding of the unsigned integer x in exactly n/8 bytes: BE16(x) is a 2-byte encoding (range 0-65535), BE32(x) is a 4-byte encoding (range 0-2^{32}−1). **|x|** denotes the byte length of byte string x.

MAC denotes HMAC-SHA3-256 throughout this document. MAC(key=k, data=d) uses k as the HMAC key and d as the HMAC data — argument labels are always explicit to prevent key/data transposition. All HKDF-based KDF functions (KDF_Root, KDF_KEX, KDF_Call) use a single HKDF invocation (Extract-then-Expand per RFC 5869) with the specified output length; the output bytes are split sequentially into the named keys at 32-byte boundaries.

KDF_Root(rk, ss) → (rk', ek): HKDF with salt = rk, ikm = ss, info = "lo-ratchet-v1", output length 64 bytes, split evenly into new root key rk' and epoch key ek.

KDF_MsgKey(ek, counter) → mk: mk = MAC(key=ek, data=0x01 ‖ BE32(counter)). The epoch key ek is static within an epoch; it does not advance per message. Forward secrecy is per-epoch (per KEM ratchet step), not per-message — see §7.2.

KDF_KEX(ikm, info) → (rk, ek): HKDF with salt = 0^{32}, ikm, info; output length 64 bytes, split evenly into root key rk and initial epoch key ek. The full info construction — including label, length-prefixed key fields, and composite-key scope — is given in §4.3 Step 3.

KDF_Call(rk, ss_eph, call_id, fp_lo, fp_hi) → (key_a, key_b, ck_call): HKDF with salt = rk, ikm = ss_eph ‖ call_id, info = "lo-call-v1" ‖ fp_lo ‖ fp_hi (74 bytes), output length 96 bytes, split into three 32-byte keys. The fingerprints in the info field bind the derived keys to the participants' identities. Role assignment (which of key_a/key_b is send vs recv) is determined by fingerprint ordering (§5.7).

KDF_CallChain(ck_call) → (key_a', key_b', ck_call'): key_a' = MAC(key=ck_call, data=[0x04]); key_b' = MAC(key=ck_call, data=[0x05]); ck_call' = MAC(key=ck_call, data=[0x06]). Each data argument is a single byte. Intra-call rekeying; the old ck_call is zeroized.

H: SHA3-256. Used for identity fingerprints and ratchet public key hashing.

2.4 Authenticated Encryption

AEAD.Enc(k, n, m, aad) → c: XChaCha20-Poly1305, 128-bit tag appended to ciphertext.

AEAD.Dec(k, n, c, aad) → m or ⊥.

Nonces for ratchet messages are counter-derived: n = 0^{20} ‖ BE32(counter) (20 zero bytes followed by the 4-byte big-endian counter, forming a 192-bit / 24-byte nonce). The nonce for the session-init first message is uniformly random (defense-in-depth; the message key is already unique by chain position).

Superscript convention: Superscripts on zero-strings denote byte counts (e.g., 0^{20} = 20 zero bytes, 0^{32} = 32 zero bytes). Superscripts on {0,1} sets denote bit counts (e.g., $\{0,1\}^{192}$ = 192-bit / 24-byte string). This dual convention is noted inline where the bit-count form appears.

2.5 Encoding

Encode(·) is an injective mapping from protocol structures to byte strings, and Decode(·) is its inverse. This ensures that AAD binding prevents structural ambiguity: distinct protocol messages always produce distinct encoded representations. For ratchet headers H = (pk_s, c_ratchet, n, pn), the optional field c_ratchet may be ⊥ (absent): Encode(H) represents the c_ratchet field as a 1-byte presence flag — 0x00 when c_ratchet = ⊥ (no ciphertext follows), 0x01 when present (followed by a 2-byte big-endian length prefix and the 1120-byte KEM ciphertext). Injectivity holds because the flag unambiguously distinguishes the two encodings; there is no length prefix of 0 for the absent case (Crypto.md §6.3 specifies the full wire format). A Tamarin/ProVerif translation should model ⊥ as a distinguished constant distinct from all byte strings.

Info string injectivity: The HKDF info constructions for KDF_KEX (§4.3 Step 3) and KDF_Call (§5.7 Step 4) are also injective, though this follows from different reasoning than Encode. KDF_KEX's info = `"lo-kex-v1" ‖ BE16(|cv|) ‖ cv ‖ BE16(|pk_IK_A|) ‖ pk_IK_A ‖ BE16(|pk_IK_B|) ‖ pk_IK_B ‖ BE16(|pk_EK|) ‖ pk_EK` is injective because the BE16 length prefixes make the concatenation unambiguous (standard length-prefix encoding). The raw `"lo-kex-v1"` prefix (9 bytes, no length prefix) is unambiguously parseable because the hard-fail version policy (§4.2 Step 3) prevents protocol version coexistence — exactly one label exists at any time, so the label length is fixed. A formal injectivity proof should either assume a fixed label length or cite the hard-fail policy as the mechanism that prevents label-length ambiguity across versions. KDF_Call's info = `"lo-call-v1" ‖ fp_lo ‖ fp_hi` (74 bytes fixed: 10 + 32 + 32) is injective because all fields have fixed sizes — distinct (fp_lo, fp_hi) pairs produce distinct info strings. These injectivity properties are load-bearing for Theorem 7 (domain separation: two sessions with different participants produce different info strings) and Theorem 1 (different sessions produce different HKDF inputs). A formal proof of Theorem 7 should include info injectivity as an explicit lemma.

---

3. Key Hierarchy

Identity Key (IK): Long-term key pair IK = (pk_IK, sk_IK), where pk_IK encodes both an X-Wing public key (for KEM) and an ML-DSA-65 public key (for signing), with the X25519 component embedded in the X-Wing key. The identity fingerprint fp_IK = H(pk_IK) serves as a compact identifier. `pk_IK = pk_IK[XWing] ‖ pk_IK[Ed25519] ‖ pk_IK[ML-DSA]` (1216 + 32 + 1952 = 3200 bytes). `sk_IK = sk_IK[XWing] ‖ sk_IK[Ed25519] ‖ sk_IK[ML-DSA]` (2432 + 32 + 32 = 2496 bytes); the 32-byte `sk_IK[Ed25519]` is the Ed25519 signing secret key (RFC 8032); the 32-byte `sk_IK[ML-DSA]` is the ML-DSA-65 seed from which the expanded signing key (4032 bytes) is derived on each signing operation — the seed is the stored form; the expanded key is not stored. The X25519 component within X-Wing is used solely for KEM; a dedicated Ed25519 keypair handles signing. A single corruption of sk_IK yields both KEM decapsulation and signing capability.

Signed Pre-Key (SPK): A medium-term X-Wing key pair (pk_SPK, sk_SPK) signed by the identity key: σ_SPK = HybridSig.Sign(sk_IK, "lo-spk-sig-v1" ‖ pk_SPK). Rotated approximately weekly; old secret keys retained for a grace period to allow delayed session inits.

One-Time Pre-Key (OPK): A single-use X-Wing key pair (pk_OPK, sk_OPK). Deleted immediately after a single decapsulation. Its presence provides enhanced forward secrecy; the protocol is functional without it.

Ephemeral Key (EK): A per-session X-Wing key pair (pk_EK, sk_EK) generated by the initiator. Serves as the initiator's initial ratchet public key.

Session Key Material: Root key rk ∈ $\{0,1\}^{256}$; send/receive epoch keys ek_s, ek_r ∈ $\{0,1\}^{256}$; message keys mk ∈ $\{0,1\}^{256}$. Epoch keys are static within an epoch and replaced on each KEM ratchet step. Message keys are single-use and zeroized after use.

sk_IK storage: sk_IK MUST be encrypted at rest. Applications use Argon2id (RFC 9106) for passphrase-based key derivation when protecting sk_IK with a user passphrase. Argon2id is not part of the LO-KEX protocol itself and is not modelled in formal analyses of §4-§6.

Lazy key validation: Sub-key validation (Ed25519 point-on-curve, ML-DSA structure checks, ML-KEM ciphertext well-formedness) is deferred to point of use, not checked at key construction or deserialization. Only byte-length validation occurs at `from_bytes`. For formal models, key validity predicates should be attached to `Sign` / `Verify` / `Encaps` / `Decaps` rules, not to key generation or deserialization rules. An invalid sub-key that is never used is never detected.

---

4. LO-KEX: Session Establishment

4.1 Pre-Key Bundle

Party B publishes:

> Bundle_B = (crypto_version, pk_IK_B, pk_SPK_B, id_SPK, σ_SPK [, pk_OPK_B, id_OPK])

where σ_SPK = HybridSig.Sign(sk_IK_B, "lo-spk-sig-v1" ‖ pk_SPK_B). The fixed label is a domain separator preventing cross-context signature reuse. **Unauthenticated indices**: σ_SPK binds pk_SPK_B but not id_SPK — an adversary controlling the bundle relay can substitute id_SPK without invalidating the signature. This is harmless: Bob uses id_SPK as a lookup key at §4.4 Step 5; a wrong id_SPK causes lookup failure, not a security breach. The same applies to id_OPK. A formal model should treat id_SPK and id_OPK as unauthenticated lookup indices whose integrity is provided by lookup failure, not by σ_SPK. Note: `crypto_version` is intentionally excluded from the signed data. Downgrade protection relies on the hard-fail version check at §4.2 Step 3 (reject any unrecognized version unconditionally), not on signature binding. If `crypto_version` were signature-bound, downgrade resistance would follow from EUF-CMA; since it is not, downgrade resistance depends on the verifier correctly rejecting unsupported versions — a local check, not a cryptographic guarantee. A Tamarin model must encode version validation as a receiver-side guard, not as a signature-binding property. See A1 (§7.5) for the full ruling. OPK fields are either both present or both absent (co-presence invariant). `id_SPK` and `id_OPK` are uint32 key identifiers; `crypto_version` is a UTF-8 string (the only supported value is `"lo-crypto-v1"`).

4.2 Bundle Verification

Party A, holding an authentic reference pk_IK_B, verifies:

1. The bundle's claimed IK equals the known pk_IK_B.
2. HybridSig.Verify(pk_IK_B, "lo-spk-sig-v1" ‖ pk_SPK_B, σ_SPK) = 1.
3. Crypto version is supported (equality check against the set of known version strings; currently `{"lo-crypto-v1"}`).
4. OPK co-presence invariant holds.

Any failure aborts. The implementation enforces that session initiation is only callable with a successfully verified bundle (type-level guarantee).

4.3 Session Initiation (Party A)

1. Generate: (pk_EK, sk_EK) ← XWing.KeyGen(). sk_EK is retained until the end of §4.3, where it initializes Alice's send ratchet secret key (see initial ratchet state below).

2. Encapsulate:

    - (c_IK, ss_IK) ← XWing.Encaps(pk_IK_B[XWing]) — where pk_IK_B[XWing] denotes the first 1216 bytes (X-Wing component) of pk_IK_B; the full 3200-byte composite key is not a valid input to XWing.Encaps
    - (c_SPK, ss_SPK) ← XWing.Encaps(pk_SPK_B)
    - If OPK present: (c_OPK, ss_OPK) ← XWing.Encaps(pk_OPK_B)

3. Derive session keys:

    - ikm = ss_IK ‖ ss_SPK [‖ ss_OPK]
    - info = "lo-kex-v1" ‖ BE16(|crypto_version|) ‖ crypto_version ‖ BE16(|pk_IK_A|) ‖ pk_IK_A ‖ BE16(|pk_IK_B|) ‖ pk_IK_B ‖ BE16(|pk_EK|) ‖ pk_EK
    - pk_IK_A and pk_IK_B are the full composite identity public keys (X-Wing + Ed25519 + ML-DSA-65, 3200 bytes each), not just the X-Wing component. The "lo-kex-v1" label is a raw 9-byte prefix without a length prefix; crypto_version and the three key fields are BE16 length-prefixed.
    - (rk, ek) ← KDF_KEX(ikm, info)

4. Construct session init: SI = (crypto_version, fp_IK_A, fp_IK_B, pk_EK, c_IK, c_SPK, id_SPK [, c_OPK, id_OPK]), where fp_IK_B = H(pk_IK_B). The `crypto_version` field is a length-prefixed string and is the first field in Encode(SI); it is therefore bound into the AEAD AAD. Including fp_IK_B directly in SI ensures the signed payload names the intended recipient explicitly — a Tamarin or ProVerif model can derive recipient binding from σ_SI alone, without reasoning about the KEM ciphertexts' implicit binding.

5. Sign session init: σ_SI ← HybridSig.Sign(sk_IK_A, "lo-kex-init-sig-v1" ‖ Encode(SI)). σ_SI proves to Bob that the session was initiated by the holder of sk_IK_A; Bob verifies it in §4.4 Step 2 before performing any KEM operations.

6. Encrypt first message:

    - mk_0 ← KDF_MsgKey(ek, 0)
    - n_0 ← uniform $\{0,1\}^{192}$ (192-bit / 24-byte uniformly random nonce; $\{0,1\}^n$ denotes an n-bit string here, unlike the byte-count superscript convention used elsewhere in this document)
    - aad_0 = "lo-dm-v1" ‖ fp_IK_A ‖ fp_IK_B ‖ Encode(SI) (fp_IK_B = H(pk_IK_B) as defined in §3; computed from pk_IK_B obtained from the verified bundle (§4.2); note fp_IK_B appears twice — once as the standalone AAD prefix and again inside Encode(SI) as SI.fp_IK_B — both occurrences are intentional: the prefix enables fast lookup without parsing the blob, and the embedded copy ties the fingerprint into the signed payload)
    - c_0 = AEAD.Enc(mk_0, n_0, m_0, aad_0)
    - Transmit (SI, σ_SI, n_0 ‖ c_0)

Note: the session-derived ek becomes the epoch key for the ratchet. Unlike the previous chain-ratchet design, the epoch key is not advanced by the first-message encryption — it is passed through unchanged. `send_count` starts at 1 so that counter 0 is not reused by the ratchet.

A's initial ratchet state: rk, ek_s = ek, ek_r = 0^{32} (initial placeholder; the all-zero value is never passed to KDF_MsgKey — the pending flag and state machine invariants ensure a KEM ratchet step always precedes first use of this field (see §5.2)), send ratchet keypair (pk_EK, sk_EK), s = 1, r = 0, pn = 0, pending = ⊥, recv_seen = ∅.

After initialization, the intermediate values ss_IK, ss_SPK, ss_OPK (if used), ikm, and the 64-byte HKDF output are zeroized. The session-derived ek is consumed by the ratchet state and must not be retained separately. The same obligation applies to §4.4.

4.4 Session Reception (Party B)

Prerequisite: Bob must have resolved `pk_IK_A` from `fp_IK_A` via a trusted binding (key directory, TOFU, or prior contact) before processing the session init. An incorrect resolution causes Step 1 (fingerprint check) or Step 2 (signature verification) to fail; the session does not establish silently with a wrong key.

1. Validate fingerprints and version: H(pk_IK_A) = SI.fp_IK_A (where H is applied to the full 3200-byte composite public key, as defined in §3); H(pk_IK_B) = SI.fp_IK_B; crypto_version supported.

2. Validate OPK co-presence: c_OPK and id_OPK are either both present or both absent; fail with InvalidData if not. Structural validation before cryptographic verification avoids unnecessary signature/KEM work on malformed messages.

3. Verify initiator signature: HybridSig.Verify(pk_IK_A, "lo-kex-init-sig-v1" ‖ Encode(SI), σ_SI) = 1; fail if 0. This proves Alice holds sk_IK_A. Executes before KEM operations so a forged or absent signature is rejected immediately. **Ordering note**: Steps 2 and 3 commute — both are read-only checks on the SessionInit payload with no state mutation or dependency on each other's outcome. The security-relevant property is "both checks pass before KEM decapsulation (Step 4)," which holds regardless of whether structural validation precedes or follows signature verification. The spec's ordering (structural first) is an efficiency preference, not a security requirement.

4. Validate `opk_sk` parameter co-presence: the caller-supplied `opk_sk` argument must be present iff `c_OPK` is present in SI (i.e., `c_OPK ≠ ⊥ iff opk_sk ≠ ⊥`); fail with InvalidData if not. This is a distinct check from Step 2 (which validates co-presence of `c_OPK` and `id_OPK` within the encoded SessionInit); Step 4's check ensures the API caller has provided the matching secret key before any KEM decapsulation is attempted. A formal model should encode this as an additional precondition of the Decrypt rule: if `c_OPK` is set in SI, a corresponding `!OPK(id, sk)` linear fact must be available.

5. Decapsulate:

    - ss_IK ← XWing.Decaps(sk_IK_B[XWing], c_IK) — where sk_IK_B[XWing] denotes the X-Wing component (first 2432 bytes) of the composite secret key sk_IK_B
    - ss_SPK ← XWing.Decaps(sk_SPK_B[id_SPK], c_SPK) — sk_SPK_B is a partial function from uint32 to secret keys; if id_SPK is not in the domain (key rotated or unknown), the lookup fails and the session is rejected
    - If c_OPK present: ss_OPK ← XWing.Decaps(sk_OPK_B[id_OPK], c_OPK)

6. Delete: if OPK was used, the caller must delete sk_OPK_B[id_OPK]. This is an irreversible action and must occur before the ratchet state is used for messaging. OPK decapsulation and deletion MUST be atomic — if two session inits reference the same OPK, at most one succeeds; the second must fail (OPK not found). Without atomicity, the TOCTOU window allows two sessions to derive identical ss_OPK, violating the forward secrecy guarantee that OPK provides (see A5, §7.5). A Tamarin model should represent OPK usage as a single rule that consumes the `!OPK(id, sk)` fact, ensuring uniqueness by construction.

7. Derive session keys (identical to §4.3 Step 3, using the resolved pk_IK_A).

8. Decrypt first message:

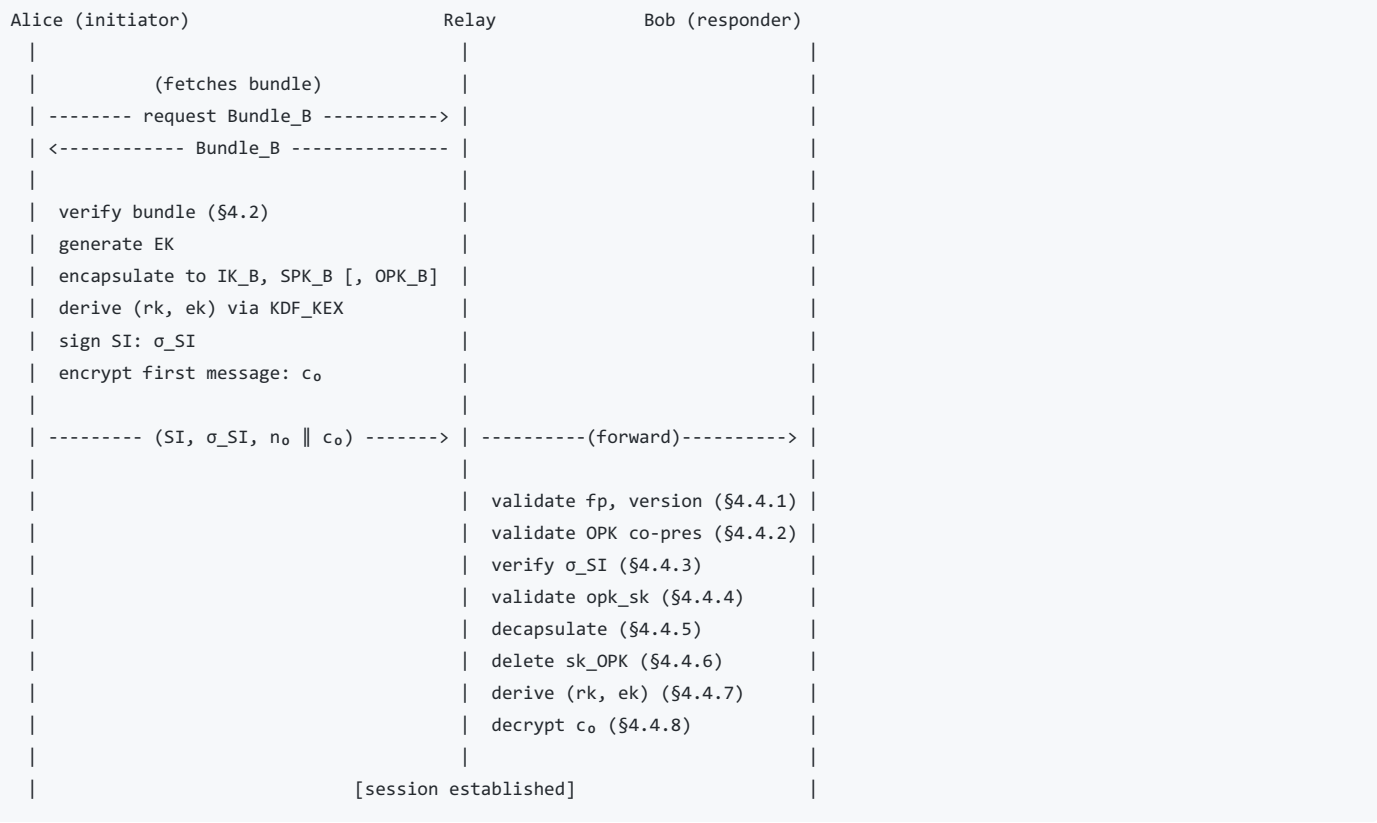
    - mk_0 ← KDF_MsgKey(ek, 0)
    - aad_0 = "lo-dm-v1" ‖ fp_IK_A ‖ fp_IK_B ‖ Encode(SI)
    - m_0 = AEAD.Dec(mk_0, n_0, c_0, aad_0); fail if ⊥.

    Formal model note: decrypt_first_message operates outside the ratchet state machine — it executes before Bob's RatchetState exists. The implementation confirms this: it is a standalone function taking (epoch_key, payload, aad), not a method on RatchetState. A Tamarin model must encode this as a separate rule from the §5.4 Decrypt rule, with a linear fact guard ensuring it fires exactly once per session. Bob's initial state has r = 1 precisely because counter 0 was consumed by this rule, not by the ratchet. A modeler who encodes Step 8 as a special case of §5.4 Decrypt would incorrectly apply recv_seen tracking, snapshot/rollback, and epoch identification to a message that predates the state machine. **Counter-0 replay note**: Because counter 0 is not inserted into recv_seen (which starts as ∅ with r = 1), a ratchet-format replay of the first message's ciphertext at n* = 0 bypasses duplicate detection — $0 \notin$ recv_seen. Protection relies on Encode disjointness (Theorem 7 sub-lemma): Encode(H) and Encode(SI) have disjoint length ranges (minimum 1,196-byte gap — see §8.6 Encode disjointness note for the size derivation), so the reconstructed AAD differs and AEAD verification fails. A model that abstracts AAD content (treating it as opaque) must add an explicit axiom preventing counter-0 reuse in the initial epoch, or faithfully encode the AAD structure to capture this protection. A second independent protection exists at the nonce level (§7.2): the session-init random nonce n_0 matches the counter-derived format 0^{20} ‖ BE32(0) with probability 2^{-160}, so even if a model abstracts AAD, faithful nonce modeling still prevents the collision. The two protections operate at different abstraction levels — a model that faithfully encodes either AAD structure or nonce construction (or both) captures the counter-0 protection.

B's initial ratchet state: rk, ek_s = 0^{32} (initial placeholder; the all-zero value is never passed to KDF_MsgKey — the pending flag and state machine invariants ensure a KEM ratchet step always precedes first use of this field (see §5.2)), ek_r = ek, pk_r = pk_EK, r = 1 (counter 0 was consumed by `decrypt_first_message` above, outside the ratchet; this is load-bearing for nonce uniqueness: the ratchet's first receive on this epoch key starts at counter ≥ 1, so the counter-derived nonce 0^{20} ‖ BE32(n) for n ≥ 1 is disjoint from counter 0 already consumed by the first message — a model mapping r to "number of ratchet Decrypt operations" would get r = 0 for a state that has already accepted one message), s = 0, pn = 0, pending = ⊤, recv_seen = ∅.

Zeroization obligation: After initialization, the intermediate values ss_IK, ss_SPK, ss_OPK (if used), ikm, and the 64-byte HKDF output are zeroized. The session-derived ek is consumed by the ratchet state and must not be retained separately.

4.5 Message Flow



4.6 Initial State After LO-KEX

Field	Alice (initiator)	Bob (responder)
rk	rk (from KDF_KEX)	rk (same)
ek_s	ek (session-derived epoch key)	0^{32} (placeholder)
ek_r	0^{32} (placeholder)	ek (session-derived epoch key)
pk_s	pk_EK	∅ (unset)
sk_s	sk_EK	∅ (unset)
pk_r	∅ (unset)	pk_EK
s	1	0
r	0	1
pn	0	0
pending	⊥	⊤
recv_seen	∅	∅
prev_ek_r	⊥ (unset)	⊥ (unset)
prev_pk_r	⊥ (unset)	⊥ (unset)
prev_recv_seen	∅	∅
local_fp	H(pk_IK_A)	H(pk_IK_B)
remote_fp	H(pk_IK_B)	H(pk_IK_A)
epoch	0	0

Alice holds the send ratchet keypair (pk_EK, sk_EK) and has already sent one message (s=1). Bob holds Alice's public key as pk_r and has pending=⊤, meaning his first send will trigger a KEM ratchet step (§5.2) to establish his own send ratchet keypair.

Initialization invariant: Fields marked 0^{32} (placeholder) and ∅ (unset) must not be used as KDF inputs before their first legitimate assignment. Specifically: Alice's ek_r = 0^{32} is never passed to KDF_MsgKey because `identify_epoch` (§5.4) returns NewEpoch when pk_r is unset, forcing a KEM ratchet step that replaces ek_r before any receive-side key derivation. Bob's ek_s = 0^{32} is never passed to KDF_MsgKey because pending = ⊤ forces a KEM ratchet step (§5.2) that replaces ek_s before any send-side key derivation. A formal model must encode these preconditions as state machine invariants — the placeholder values are unreachable, not merely unused.

Init preconditions: `init_alice` and `init_bob` reject all-zero rk or ek values (constant-time check). All-zero KEX output indicates a degenerate HKDF result — cryptographically implausible but structurally guarded. A formal model should encode this as a precondition on the `SessionEstablished` rule: the KDF_KEX output must be non-trivial.

Fork prevention: Serialization of Σ is a destructive read — `to_bytes()` consumes Σ and returns the serialized blob. The in-memory state is zeroized upon serialization. At any point in time, at most one live copy of a session's ratchet state exists. Without this invariant, an application could serialize, continue encrypting, then restore from the serialized copy, causing catastrophic (epoch_key, nonce) reuse. Combined with the epoch anti-rollback counter (§5.1), this prevents nonce reuse via state forking at both the language level (ownership consumption) and the storage level (epoch monotonicity). A Tamarin model should encode RatchetState as a linear fact — consumed by serialization and re-created by deserialization with epoch' > epoch.

---

5. LO-Ratchet

5.1 State

Σ = (rk, ek_s, ek_r, pk_s, sk_s, pk_r, prev_ek_r, prev_pk_r, s, r, pn, pending, recv_seen, prev_recv_seen, local_fp, remote_fp, epoch)

- rk: root key
- ek_s, ek_r: send and receive epoch keys (static within an epoch; replaced on each KEM ratchet step)
- (pk_s, sk_s): current send ratchet keypair (may be unset)
- pk_r: current receive ratchet public key (may be unset)
- prev_ek_r: previous receive epoch key (retained for one epoch to handle late-arriving messages from the prior epoch; overwritten on the next KEM ratchet step — the old value is zeroized via `ZeroizeOnDrop` when replaced)
- prev_pk_r: previous receive ratchet public key (identifies which epoch a late-arriving message belongs to)

- s: send counter; r: receive counter (high-water mark for current epoch — the supremum of `{n + 1 : n ∈ successfully_decrypted}`, updated via `r ← max(r, n*
    - 1) `in §5.4 Step 6; a single out-of-order message can advance r from 0 to an arbitrary value while |recv_seen| grows by exactly 1; a model using r as a message count will produce incorrect reachability results). **Derivation relationship**: r is not independent of recv_seen — it is exactly` max(recv_seen) + 1 `when` recv_seen ≠ ∅ `, and  0 `when` recv_seen = ∅ `(modulo the initial r = 1 for Bob after session establishment, where recv_seen = ∅ but r = 1 due to counter-0 retirement §4.4 Step 8). A model that treats r and recv_seen as independent terms can represent unreachable states (e.g., r = 5 with recv_seen = {17}). To avoid this, either derive r from recv_seen or add an explicit coupling invariant:` recv_seen = ∅ ⇒ r ∈ {0, 1} and recv_seen ≠ ∅ ⇒ r = max(recv_seen) + 1`; pn: previous send chain length (set to s at the time of a KEM ratchet step — records how many messages were sent in the previous send epoch). **Receive-side note**: pn is included in the ratchet header and bound into the AEAD AAD (§5.3 Step 5), but has no cryptographic function on the receive side — no Decrypt operation reads or branches on pn's value. In Signal's Double Ratchet, pn tells the receiver how many skip keys to pre-derive; in LO-Ratchet's counter-mode design this is unnecessary. A formal model can treat pn as an opaque integer in the header tuple, contributing to AAD integrity but not to state transitions or key derivation
- pending ∈ {⊤, ⊥}: ⊤ when a new pk_r has been received but the send-side KEM ratchet step has not yet been performed. pending = ⊤ does not prevent sending; it triggers a KEM ratchet step (§5.2) before the next send (§5.3 Step 2)
- recv_seen: set of u32 counters for messages successfully decrypted in the current receive epoch. Bounded at MAX_RECV_SEEN (65536) entries. Resets on KEM ratchet step. **Modeling note**: MAX_RECV_SEEN is a pure resource bound (DoS mitigation), not a security parameter — it does not appear in any theorem's security loss, any freshness predicate, or any reduction. A formal model can safely use an unbounded recv_seen set without invalidating any theorem in §8.6. The bound's only formal implication is the ChainExhausted terminal state at §5.4 Step 6, which is a liveness property, not a safety/secrecy property. See §9.4 for the bounded analysis as a standalone problem.
- prev_recv_seen: set of u32 counters for messages successfully decrypted from the previous receive epoch. Resets on KEM ratchet step (when prev_ek_r is overwritten).
- local_fp: identity fingerprint of the local party (H(pk_IK_local)). Immutable after session init. Used in AAD construction (§5.3 Step 5) as fp_sender (on send) or fp_recipient (on receive). Binding local_fp into AAD prevents cross-session replay: a ciphertext produced under one session's keys cannot validate under a different session with different participants.
- remote_fp: identity fingerprint of the remote party (H(pk_IK_remote)). Immutable after session init. local_fp ≠ remote_fp is enforced at construction (self-messaging is rejected).
- epoch: u64 monotonic anti-rollback counter. Incremented each time the state is serialized (via `to_bytes()`). On deserialization, `from_bytes_with_min_epoch(blob, min_epoch)` rejects states where epoch ≤ min_epoch (strictly greater required). **Terminology note**: "epoch" has two distinct meanings in this specification. The **storage epoch** (this field) is a persistence-layer counter advanced by serialization. A **cryptographic epoch** (used in §8.3 F1-F4, Theorems 3-5) refers to a KEM ratchet step boundary — a new cryptographic epoch begins each time a KEM ratchet step replaces the epoch key via KDF_Root. These are independent: multiple serializations may occur within a single cryptographic epoch, and a KEM ratchet step does not advance the storage epoch counter. A formal model needs both notions: cryptographic epochs for freshness predicates, storage epochs for anti-rollback. This prevents storage-layer replay attacks where an adversary substitutes an older serialized state — or the same serialized state — to rewind or freeze the ratchet. **Trust anchor**: the `min_epoch` value must be stored in a location with integrity guarantees independent of the ratchet blob — if an adversary who can substitute the blob can also substitute `min_epoch`, the anti-rollback mechanism is defeated. The application is responsible for providing a trustworthy `min_epoch` (e.g., via a separate authenticated store or monotonic counter). A formal model should treat `min_epoch` as an oracle input with an integrity assumption.

Spec	Rust field
rk	`root_key`
ek_s	`send_epoch_key`
ek_r	`recv_epoch_key`
pk_s, sk_s	`send_ratchet_pk`, `send_ratchet_sk`
pk_r	`recv_ratchet_pk`
prev_ek_r	`prev_recv_epoch_key`
prev_pk_r	`prev_recv_ratchet_pk`
s	`send_count`
r	`recv_count`
pn	`prev_send_count`
pending	`ratchet_pending`
recv_seen	`recv_seen`
prev_recv_seen	`prev_recv_seen`
local_fp	`local_fp`
remote_fp	`remote_fp`
epoch	`epoch`

State invariants (hold at all quiescent points between protocol operations):

- (a) pending = ⊤ implies pk_r is set.
- (b) s > 0 implies (pk_s, sk_s) are set.
- (c) r > 0 implies pk_r is set. (One-directional: pk_r may be set with r = 0 immediately after a KEM ratchet step resets recv_count.)
- (d) s = 0 ∧ sk_s set is unreachable.
- (e) s = 0 ∧ pending = ⊥ ∧ pk_r set ∧ sk_s unset is unreachable.
- (f) prev_ek_r is set iff prev_pk_r is set (co-presence).
- (g) |recv_seen| ≤ MAX_RECV_SEEN; |prev_recv_seen| ≤ MAX_RECV_SEEN. Note: the runtime bound allows recv_seen to reach exactly MAX_RECV_SEEN entries (the check-before-insert pattern at §5.4 Step 6 passes at MAX_RECV_SEEN − 1, then the insert produces MAX_RECV_SEEN). However, a state with MAX_RECV_SEEN entries cannot be serialized — serialization rejects at |recv_seen| ≥ MAX_RECV_SEEN. The serialization invariant is therefore |recv_seen| < MAX_RECV_SEEN. Models reasoning about serialization should use the strict bound.
- (h) local_fp ≠ remote_fp (self-messaging rejected at construction). Additionally, local_fp ≠ 0^{32} and remote_fp ≠ 0^{32} (all-zero fingerprint rejected at construction; constant-time check). Under SHA3-256 modeled as a random oracle, H(pk_IK) = 0^{32} has probability 2^{-256} — cryptographically impossible but structurally guarded, consistent with the philosophy behind (j)-(m). A symbolic model that does not inherently guarantee non-zero hash outputs should include this as an axiom on SessionEstablished.
- (i) epoch is monotonically non-decreasing across serialization/deserialization cycles.
- (j) rk ≠ 0^{32} (dead sessions are not restorable; constant-time check).
- (k) ek_s ≠ 0^{32} when s > 0 ∧ pending = ⊥ (active send epoch has non-degenerate key). When pending = ⊤, ek_s may be stale (all-zero or from a prior epoch) because the next Send triggers a KEM ratchet step (§5.2) that replaces ek_s before any key derivation uses it.
- (l) ek_r ≠ 0^{32} when r > 0 ∧ pending = ⊥ (active receive epoch has non-degenerate key).
- (m) prev_ek_r ≠ 0^{32} when prev_ek_r is set (retained previous-epoch key is non-degenerate).

  Invariants (j)-(m) are maintained by a two-part inductive argument. The **base case** is mechanically enforced: `init_alice` / `init_bob` reject all-zero rk or ek values via constant-time checks (§4.6 init preconditions). The **inductive step** is probabilistic: `KDF_Root(rk, ss)` produces a 64-byte HKDF output, and under the random oracle model each 32-byte component (rk', ek) equals 0^{32} with probability 2^{-256} per KEM ratchet step — non-degenerate with overwhelming probability but not structurally guaranteed. A Tamarin model can sidestep this (the symbolic random oracle never produces the zero constant). A CryptoVerif or game-based reduction should account for the 2^{-256} loss per ratchet step in the security bound.
- (n) s, r, pn < 2^{32}−1 (exhausted counters cannot be deserialized).
- (o) epoch < 2^{64}−1 (epoch at maximum cannot be re-serialized; `to_bytes` stores epoch+1, overflow returns ChainExhausted). `from_bytes` rejects stored epoch = 2^{64}−1. States with epoch = 2^{64}−2 are accepted by `from_bytes` but `can_serialize()` returns false — the session is usable for encrypt/decrypt but not persistable.
- (p) All entries in recv_seen are strictly ascending and each entry < r (high-water mark consistency). Note: the ascending-order property is a serialization invariant — at runtime, recv_seen is an unordered set (HashSet). The runtime invariant is the weaker ∀ n ∈ recv_seen: n < r ∧ |recv_seen| ≤ MAX_RECV_SEEN. Models reasoning about intermediate states between serialization points should use the runtime form.
- (q) prev_recv_seen is empty when prev_ek_r is unset (no orphaned duplicate-detection state).
- (r) X-Wing secret key's X25519 component ≠ 0^{32} when present (degenerate DH scalar rejected; constant-time check).
- (s) No trailing bytes after the last serialized field (strict parsing; prevents length-extension ambiguity).

Base case: The §4.6 initial states satisfy all invariants (a)-(s) by construction. Several hold non-obviously via vacuous antecedent satisfaction: (c) r > 0 — Alice has r = 0 (vacuous), Bob has r = 1 with pk_r = pk_EK (satisfied). (d) s = 0 ∧ sk_s set — Bob has s = 0 and sk_s = ∅ (satisfied), Alice has s = 1 (vacuous). (k) ek_s ≠ 0^{32} when s > 0 ∧ pending = ⊥ — Alice has s = 1, pending = ⊥, ek_s = ek (non-zero per init precondition, satisfied). Bob has s = 0 (vacuous). (l) ek_r ≠ 0^{32} when r > 0 ∧ pending = ⊥ — Bob has pending = ⊤ (antecedent false, vacuous). The antecedent designs of (c), (d), (k), and (l) specifically accommodate the asymmetric Alice/Bob initialization states.

These correspond to the validation checks enforced by deserialization and are suitable as reachability lemma targets in a Tamarin model.

Mutual exclusion: All operations on a session state Σ (Encrypt, Decrypt, DeriveCallKeys, serialization) are assumed to execute under mutual exclusion. Concurrent access to the same Σ is undefined behavior. The implementation enforces this via Rust's `&mut self` borrow checker — all mutating operations require exclusive access. In Tamarin, mutual exclusion is enforced by construction: the linear `RatchetState(sid, ...)` fact is consumed and re-produced by each rule, preventing a second rule from firing on the same session concurrently. Other frameworks (e.g., concurrent process calculi, CryptoVerif with parallel oracles) must encode this as an explicit assumption. Without serialization, the snapshot/rollback mechanism in Decrypt (§5.4) is unsound — it assumes send-side fields are not being concurrently mutated by Encrypt.

5.2 KEM Ratchet Step (Send)

Triggered when pk_s is unset or pending = ⊤:

1. (pk_s', sk_s') ← XWing.KeyGen()
2. (c_ratchet, ss) ← XWing.Encaps(pk_r)
3. (rk', ek_s') ← KDF_Root(rk, ss)
4. pn ← s; s ← 0; rk ← rk'; ek_s ← ek_s'; (pk_s, sk_s) ← (pk_s', sk_s'); pending ← ⊥
5. Return c_ratchet

5.3 Send

Encrypt(Σ, m):

0. If rk = 0^{32}: fail (InvalidData). (Liveness guard: all-zero root key indicates a dead session — post session-fatal error or post-reset. Constant-time comparison.)
1. If s = 2^{32}−1: fail (ChainExhausted). (This guard also ensures Step 7's `s ← s
   - 1` does not overflow u32 — with s ≤ 2^{32}−2 at this point, s + 1 ≤ 2^{32}−1.)**Non-terminal failure**: ChainExhausted is a per-epoch non-terminal failure — the session remains alive (rk ≠ 0^{32}) and a subsequent KEM ratchet step (triggered by the next direction change) resets the counter, enabling further operations. This is distinct from session-fatal zeroization (rk ← 0^{32}, permanent, Dead state below) and from call chain exhaustion (§5.7, terminal — all

call keys zeroized). **Send-terminal caveat**: if s = $2^{32}-1$ and pending = ⊤, the KEM ratchet step (Step 2) that would reset s never executes — Step 1 rejects before Step 2 runs. The sender is permanently unable to send (no code path resets s without a successful Send), though the session remains alive for receiving. A Tamarin model's Send rule with s = $2^{32}-1$ has no successor state for the send direction regardless of the pending flag. A Tamarin model's ChainExhausted rule must re-produce the RatchetState fact unchanged (state rollback), not consume SessionAlive — the session can recover via epoch transition (unless the send counter has saturated as described above).

2. If pk_s unset or pending = ⊤: c_ratchet ← KEM_Ratchet_Step_Send(Σ); else c_ratchet ← ⊥.
3. mk ← KDF_MsgKey(ek_s, s).
4. H = (pk_s, c_ratchet, n, pn). (n equals the send counter s at time of header construction.)
5. aad = "lo-dm-v1" ‖ fp_sender ‖ fp_recipient ‖ Encode(H).
6. c ← AEAD.Enc(mk, 0^{20}‖BE32(n), m, aad).
7. s ← s + 1. Return (H, c).

Atomicity: State updates are atomic - Σ transitions to Σ' only upon successful completion of all steps. Failure before step 6 (AEAD.Enc) leaves the state at Σ. (The KEM ratchet step in step 2 (§5.2) is itself atomic: key generation and encapsulation — the only fallible operations — execute before any state field is mutated.) Failure at step 6 is session-fatal: all key material in Σ is zeroized. The KEM ratchet step (step 2) may have already mutated rk, ek_s, pk_s, and cleared the pending flag — these mutations cannot be safely unwound, so the defense-in-depth response is full zeroization to prevent any possibility of nonce reuse from retry attempts. **Formal model simplification**: In the standard cryptographic model, AEAD.Enc is a deterministic total function — given a valid key and nonce, it always produces output. Nonce uniqueness is guaranteed by construction (§7.2), and key freshness is guaranteed by the KDF chain. The session-fatal path is therefore unreachable in any standard-model formal analysis; it exists purely as implementation defense-in-depth against memory/allocation failures. A Tamarin/ProVerif model can safely treat the Send rule as deterministic (always succeeds after a successful KEM ratchet step), eliminating the Send-side failure branch from the atomicity asymmetry discussion (§5.3) and halving the Send rule's branching factor.

Dead state: After session-fatal zeroization, rk = 0^{32} and all subsequent Encrypt/Decrypt calls fail at Step 0 (liveness guard). The session is permanently unusable; a new LO-KEX session must be established. For Tamarin/ProVerif modeling, this corresponds to consuming a `SessionAlive(sid)` persistent fact on session-fatal error — no further Send or Receive rules can fire for that session.

Atomicity asymmetry (Send vs Receive): The atomicity mechanisms differ between Encrypt and Decrypt. Decrypt (§5.4) uses true snapshot/rollback of nine receive-side fields — on AEAD failure, the state is fully restored and the session continues. Encrypt uses terminal zeroization: if AEAD.Enc fails after the KEM ratchet step (Step 2) has already mutated (rk, ek_s, pk_s, pn, pending), those mutations cannot be safely unwound, so the session is destroyed rather than rolled back. A Tamarin model must encode this asymmetry: the Receive rule has two outcomes (success → state advances, failure → state unchanged), while the Send rule after a KEM ratchet step has two outcomes (success → state advances, failure → SessionAlive(sid) consumed, session terminated). Assuming symmetric rollback for both operations produces an incorrect model.

5.4 Receive

Decrypt(Σ, H, c), where H = (pk_s*, c_ratchet*, n*, pn*):

0. If rk = 0^{32}: fail (InvalidData). (Liveness guard: all-zero root key indicates a dead session. Constant-time comparison.)

1. If n* = $2^{32}-1$: fail (ChainExhausted). (Checked before any state-mutating step to prevent state mutation on a crafted counter value. This guard also ensures Step 6's `r ← max(r, n* + 1)` does not overflow u32 — with n* ≤ $2^{32}-2$ at this point, n* + 1 ≤ $2^{32}-1$. **Ordering note**: The spec places this check before epoch identification (Step 2), but the two steps commute — epoch identification is a read-only classification of pk_s* against {prev_pk_r, pk_r} with no state mutation. The security-relevant property is "counter check before any state mutation," which holds regardless of whether the counter check precedes or follows epoch identification.)

2. Identify epoch (**priority matching** — previous-epoch check takes precedence; branches are evaluated in order, not as independent predicates):

    - If pk_s* = prev_pk_r: **previous epoch** → goto step 3a.
    - Else if pk_s* = pk_r: **current epoch** → goto step 3b.
    - Else: **new epoch (KEM ratchet needed)** → goto step 3c.

    Epoch classification is determined entirely by pk_s* matching — the c_ratchet* field is not examined until Step 3c (NewEpoch only). A CurrentEpoch or PreviousEpoch message may carry an arbitrary c_ratchet* value; it is silently ignored. A formal model must not use c_ratchet* ≠ ⊥ as a precondition for the NewEpoch rule or c_ratchet* = ⊥ for Current/PreviousEpoch rules — doing so would incorrectly restrict the adversary's message space and produce an unsound model.

3a. Previous epoch:

- If prev_ek_r is unset: fail (no previous epoch key retained).
- mk ← KDF_MsgKey(prev_ek_r, n*); goto step 4 (with seen_set = prev_recv_seen).

3b. Current epoch:

- mk ← KDF_MsgKey(ek_r, n*); goto step 4 (with seen_set = recv_seen).

3c. New epoch (KEM ratchet step):

- **Precondition**: c_ratchet* ≠ ⊥ (a new-epoch message must contain a KEM ciphertext). If c_ratchet* = ⊥: fail (InvalidData). This guard must be evaluated before any state mutation below.
- **Precondition**: sk_s must be set. If sk_s = ∅: fail (InvalidData). A legitimate new-epoch message is unreachable unless sk_s is set — a new-epoch message from the remote party means the remote party performed a KEM ratchet step that encapsulated to the local party's pk_s, which only exists after the local party has sent at least once (triggering §5.2). An adversary can craft a message with an unknown pk_s* that routes to Step 3c, but without this guard the Decaps call would operate on uninitialized state. For Tamarin: this translates to a reachability lemma — NewEpochRecv(sid) is only reachable after Send(sid) has fired.
- If pk_r is set: prev_ek_r ← ek_r; prev_pk_r ← pk_r; prev_recv_seen ← recv_seen. (Conditional: on the very first receive after the decryptor's init, pk_r may be unset — saving it would produce an all-zero prev_ek_r that deserialization rejects.) Else: prev_ek_r ← ⊥; prev_pk_r ← ⊥; prev_recv_seen ← ∅. (Clear previous-epoch state — no valid previous epoch exists yet.) **Ghost variable note**: Invariant (p) bounds recv_seen entries by r, but this assignment moves recv_seen → prev_recv_seen while r is subsequently reset to 0 (below). The old r value that bounded those entries is overwritten; no `prev_r` field exists in Σ. A formal model proving membership properties about prev_recv_seen must introduce an auxiliary ghost variable — but **not** `prev_r` (the receiver's high-water mark at transition time). The receiver's r at transition only bounds counters seen so far; late-arriving PreviousEpoch messages can have counter values beyond the receiver's high-water mark at transition (e.g., sender sent counters 0-9, receiver saw 0-4 before transition, then counter 7 arrives late — 7 ≥ prev_r = 5). The correct ghost variable is `prev_send_bound`, capturing pn* from the first NewEpoch message's header (the sender's send count at the time of their epoch transition). The invariant is: ∀ n ∈ prev_recv_seen: n < prev_send_bound. This holds because AEAD authentication ensures only authentic messages enter the set, and no authentic message has counter ≥ the sender's s at epoch transition. Equivalently, this can be expressed as a correspondence property: every n ∈ prev_recv_seen corresponds to a Sent(sid, prev_epoch, n) action by the peer. The pn header field — which "has no cryptographic function on the receive side" (§5.1) — is exactly the ghost variable needed for prev_recv_seen invariant reasoning. Note that pn is publicly observable (§8.1: "pn (number of messages sent in the previous epoch)" is transmitted in cleartext and emitted as Out(H)); prev_send_bound is therefore already in the adversary's knowledge and need not be modeled as an auxiliary fact — a modeler can extract it directly from the public header of the first NewEpoch message.
- (rk', ek_r') ← KDF_Root(rk, XWing.Decaps(sk_s, c_ratchet*)). (sk_s is the decryptor's current send ratchet secret key — the remote party encapsulated to the

corresponding pk_s.)
- rk ← rk'; ek_r ← ek_r'; pk_r ← pk_s*; r ← 0; pending ← ⊤; recv_seen ← ∅.
- mk ← KDF_MsgKey(ek_r, n*); goto step 4 (with seen_set = recv_seen).

4. aad = "lo-dm-v1" ‖ fp_sender ‖ fp_recipient ‖ Encode(H). m ← AEAD.Dec(mk, 0^{20}‖BE32(n*), c, aad). If ⊥: fail (restore Σ from snapshot).
5. If n* ∈ seen_set: fail (DuplicateMessage); the successfully-decrypted plaintext from Step 4 is discarded (not returned to the caller). (Duplicate detection is post-AEAD: the full key derivation and decryption path is traversed before checking recv_seen, preventing timing side channels that would leak epoch membership of replayed messages. The epoch identification branch (Step 2) is determined by public header fields (pk_s*) and therefore does not constitute a timing side channel — the code path variation between current-epoch, previous-epoch, and new-epoch processing is observable only to the extent that the header already reveals the epoch type.) **Modeling note**: recv_seen provides replay resistance as an application-layer stateful check, orthogonal to AEAD cryptographic security. AEAD is stateless and deterministic: a replayed valid ciphertext with the same (key, nonce, ciphertext, AAD) always decrypts successfully — AEAD provides no replay resistance by itself. In the standard IND-CPA/INT-CTXT game, the adversary's freshness condition prevents submitting the same ciphertext twice, so replay is outside the AEAD security model. A Tamarin model's Decrypt rule therefore has two distinct failure modes: (a) *cryptographic failure* — AEAD tag invalid (adversary cannot trigger without the message key), and (b) *application-layer failure* — recv_seen membership (adversary can trigger by replaying a legitimate ciphertext; AEAD succeeds but the plaintext is discarded). These must be modeled as separate failure branches with different adversary capabilities.
6. If |seen_set| ≥ MAX_RECV_SEEN: fail (ChainExhausted). seen_set ← seen_set ∪ {n*}. If epoch is CurrentEpoch or NewEpoch: r ← max(r, n* + 1). (PreviousEpoch messages do not update r — they belong to a prior receive epoch, and updating the current-epoch high-water mark would corrupt recv_count.) **Recv-side recovery asymmetry**: ChainExhausted at Step 6 is terminal for the current receive epoch, not for the session (rk ≠ 0^{32}). Recovery (recv_seen reset to ∅) occurs only when the remote party sends from a new epoch, triggering NewEpoch at Step 3c. Local sends do not help — a send-side KEM ratchet step (§5.2) resets s and generates fresh send keys but does not reset recv_seen. Recovery is therefore remote-party-dependent: a Dolev-Yao adversary who controls delivery can indefinitely delay the remote party's NewEpoch message, keeping the local party permanently receive-blocked for that epoch. This affects §9.4's liveness analysis — liveness under recv-side exhaustion requires a fairness assumption on the remote party (eventually sends from a new epoch). Return m.

Atomicity and snapshot/rollback: State updates are atomic — Σ transitions to Σ' only upon successful completion of all steps. On failure, the state remains Σ. The implementation achieves this via snapshot/rollback of the nine receive-side fields that Decrypt may mutate: {rk, ek_r, r, pk_r, pending, prev_ek_r, prev_pk_r, recv_seen, prev_recv_seen}. These are snapshotted before Step 2 (epoch identification). On any failure after the snapshot (AEAD failure at Step 4, DuplicateMessage at Step 5, ChainExhausted at Step 6), all nine fields are restored from the snapshot. **Post-AEAD NewEpoch rollback**: for a NewEpoch message, Steps 3c mutates rk, ek_r, pk_r, pending, prev_ek_r, prev_pk_r, and recv_seen before AEAD decryption at Step 4. If Step 4 succeeds but Step 5 (DuplicateMessage) or Step 6 (ChainExhausted) fails, the entire KEM ratchet step — including root key advancement and epoch key derivation — is rolled back. A Tamarin model that encodes the KEM ratchet and duplicate/exhaustion checks as separate sequential rules would need a compensating rollback rule for these post-AEAD failures. The cleaner encoding is a single Decrypt_NewEpoch_Success rule with AEAD success ∧ n* ∉ recv_seen ∧ |recv_seen| < MAX_RECV_SEEN as conjunctive preconditions, producing the fully updated state only when all three conditions hold. **pk_r bifurcation**: Step 3c branches on whether pk_r is set: if set, the current epoch state is saved into prev_* fields (enabling subsequent PreviousEpoch decryption); if unset (first receive after init), prev_* is cleared (no valid previous epoch exists). These produce semantically different successor states — one with a usable previous epoch, one without. A Tamarin model should encode two Decrypt_NewEpoch_Success rules (pk_r set vs unset), or include pk_r presence as an additional branching precondition. A model that does not distinguish these cases will either incorrectly enable PreviousEpoch decryption before the first full epoch cycle, or lose reachability for PreviousEpoch processing entirely. Send-side state (ek_s, pk_s, sk_s, s, pn) is never mutated by Decrypt and is not snapshotted. The remaining fields — local_fp, remote_fp, and epoch — are immutable during protocol operations (local_fp and remote_fp are set at construction per invariant (h); epoch is advanced only by serialization, not by Encrypt or Decrypt). The nine snapshotted fields plus the five unmodified send-side fields plus the three immutable fields partition all 17 fields of Σ (per §5.1). For Tamarin/ProVerif modeling: the snapshot corresponds to a persistent fact `RecvSnapshot(sid, ...)` created before branching and consumed on either success or failure to restore state.

NewEpoch rollback re-derivability: Rollback after a failed NewEpoch step (3c) is safe because XWing.Decaps is deterministic: re-processing the same c_ratchet* after rollback re-derives the identical KEM shared secret, and therefore the identical (rk', ek_r') from KDF_Root. A message that causes AEAD failure (e.g., network corruption of the ciphertext but not the header) does not permanently block the epoch transition — the next valid message from the same epoch, carrying the same c_ratchet*, will re-trigger Step 3c and succeed. A Tamarin model's failure rule for NewEpoch must not consume the KEM ciphertext from the adversary's knowledge — it remains available for re-delivery. The success rule consumes the linear RatchetState fact and produces a new one with updated fields; the failure rule consumes and re-produces the same RatchetState.

Out-of-order delivery: Counter-mode key derivation allows any message within an epoch to be decrypted in O(1) regardless of arrival order. The recv_seen set tracks which counters have been successfully decrypted; any counter not in the set can be derived directly from the epoch key. No skip cache or sequential chain advancement is needed. **NewEpoch replay reclassification**: the first delivery of a NewEpoch message fires Step 3c (KEM ratchet step, fresh recv_seen = ∅), and the epoch-transitioning message is structurally un-duplicatable within that rule (it enters an empty set). If the adversary replays the same message, identify_epoch (Step 2) now classifies pk_s* as matching the updated pk_r (assigned in Step 3c), routing the replay to the CurrentEpoch path (Step 3b), where recv_seen catches the duplicate at Step 5. A Tamarin model must encode two distinct rules for what appears to be the same message: the first delivery fires Decrypt_NewEpoch (KEM ratchet + fresh recv_seen), and a replay fires Decrypt_CurrentEpoch (duplicate detection via recv_seen). Replay protection for NewEpoch messages works via epoch reclassification, not within-rule detection.

Pending flag state machine: The `pending` flag transitions are described across §4.6, §5.2, §5.3, and §5.4. The following state machine summarizes all transitions and maps directly to Tamarin multiset-rewriting rules:

```
States: {Idle, PendingSend}
Transitions:
  (Idle,        Send)          → Idle          [no KEM step needed]
  (PendingSend, Send)          → Idle          [KEM ratchet step (§5.2), pending ← ⊥]
  (*,           Recv_NewEpoch) → PendingSend   [pending ← ⊤]
  (*,           Recv_Current)  → unchanged
  (*,           Recv_Previous) → unchanged
```

`pending` cannot transition from PendingSend to PendingSend without an intermediate Send — a second Recv_NewEpoch while pending = ⊤ triggers a new KEM ratchet step on the receive side (§5.4 Step 3c) but sets pending ← ⊤ again, which leaves pending unchanged (already ⊤). The KEM ratchet step itself is not idempotent — it executes a full Decaps, KDF_Root, root key advancement, epoch key replacement, prev_ek_r rotation, and recv_seen reset, producing a distinct successor state.

5.5 Ratchet Message Flow

The following shows a typical exchange with a KEM ratchet step. Alice has just completed LO-KEX (s=1, holds pk_EK/sk_EK). Bob's first send triggers a KEM ratchet step.

```
Alice                                          Bob
  |                                             |
  |  state: s=1, pk_s=pk_EK, sk_s=sk_EK         |  state: r=1, pk_r=pk_EK, pending=T
  |                                             |
  |  encrypt(m₁): mk₁←KDF_MsgKey(ek_s, 1), s←2  |
  | ----------- H₁=(pk_EK, ⊥, 1, 0), c₁ ------> |
  |                                             |  decrypt: pk_EK = pk_r → current epoch
  |                                             |  mk₁←KDF_MsgKey(ek_r, 1), r←2
  |                                             |
  |                                             |  encrypt(m₂): pending=T → KEM ratchet
  |                                             |    (pk_B, sk_B) ← KeyGen()
  |                                             |    (c_r, ss) ← Encaps(pk_EK)
  |                                             |    (rk', ek_s') ← KDF_Root(rk, ss)
  |                                             |    pn←s=0, s←0, pending←⊥
  |                                             |    mk₂←KDF_MsgKey(ek_s', 0), s←1
  | <---------- H₂=(pk_B, c_r, 0, 0), c₂ ------ |
  |                                             |
  |  decrypt: pk_B ≠ pk_r → new epoch (ratchet step) |
  |    ss ← Decaps(sk_EK, c_r)                  |
  |    (rk', ek_r') ← KDF_Root(rk, ss)          |
  |    (pk_r unset → no prev save), pk_r←pk_B, r←0, pending←T |
  |    mk₂←KDF_MsgKey(ek_r', 0), r←1            |
  |                                             |
  |  encrypt(m₃): pending=T → KEM ratchet       |
  |    (pk_A', sk_A') ← KeyGen()                |
  |    (c_r', ss') ← Encaps(pk_B)               |
  |    pn←s=2, s←0, pending←⊥                   |
  |    mk₃←KDF_MsgKey(ek_s', 0), s←1            |
  | --------- H₃=(pk_A', c_r', 0, 2), c₃ ------> |
  |                                             |
```

5.6 KEM Ratchet State Transition

State before and after a KEM ratchet step on send (§5.2), assuming the sender previously had `s` messages on the old chain:

Field	Before	After
rk	rk	rk' = KDF_Root(rk, ss).root
ek_s	ek_s (old epoch)	ek_s' = KDF_Root(rk, ss).epoch
pk_s	pk_s (old)	pk_s' (freshly generated)
sk_s	sk_s (old)	sk_s' (freshly generated)
s	s (old epoch count)	0
pn	pn (old)	s (saved old epoch count)
pending	⊤	⊥
rk, ek_r, pk_r, r	unchanged	unchanged

The KEM shared secret ss is derived via `Encaps(pk_r)`. The old `(pk_s, sk_s)` and old `ek_s` are zeroized after the transition.

5.7 Call Key Derivation

Voice call key material is derived from the ratchet root key and an ephemeral KEM shared secret exchanged during call signaling. The ephemeral KEM provides forward secrecy independent of the ratchet state.

Call Setup:

1. Call initiator generates call_id ← uniform $\{0,1\}^{128}$ (a uniformly random 128-bit / 16-byte value; $\{0,1\}^n$ denotes an n-bit string, consistent with §4.3 Step 6 notation) and (pk_eph, sk_eph) ← XWing.KeyGen(). Sends (call_id, pk_eph) to the peer via a ratchet-encrypted signaling message.

2. Peer computes (c_eph, ss_eph) ← XWing.Encaps(pk_eph). Sends c_eph via a ratchet-encrypted signaling message.

3. Initiator computes ss_eph ← XWing.Decaps(sk_eph, c_eph). sk_eph is zeroized.

4. Both parties derive call keys: (key_a, key_b, ck_call) ← KDF_Call(rk, ss_eph, call_id, fp_lo, fp_hi). ss_eph is zeroized. This operation is a pure function of rk and the call parameters; Σ is unchanged (rk is read, not advanced). The implementation confirms: `derive_call_keys(&self, ...)` takes an immutable reference. In Tamarin, the `DeriveCallKeys` rule must consume and re-produce an identical `RatchetState(sid, ...)` fact (read-only access in linear logic), unlike Send/Decrypt which produce a modified copy.

   Call preconditions (checked before KDF_Call, constant-time where noted):

   - rk ≠ 0^{32} (dead session guard; constant-time). Mirrors invariant (j).
   - ss_eph ≠ 0^{32} (degenerate KEM output; constant-time). Consistent with the invariants (j)-(m) philosophy: cryptographically implausible under X-Wing but structurally guarded.
   - call_id ≠ 0^{16} (degenerate call identifier). A zero call_id could collide with a hypothetical default or uninitialized value; this guard is defense-in-depth.
   - local_fp ≠ remote_fp (self-call rejected). Re-asserts invariant (h) at the call layer; load-bearing for role assignment correctness (Step 5) — if fp_lo = fp_hi, role assignment is ambiguous.

   A formal model's `DeriveCallKeys` rule should include these as preconditions, parallel to the `SessionEstablished` preconditions in §4.6.

5. Role assignment: the party with the lexicographically lower identity fingerprint uses key_a as their send key and key_b as their recv key; the other party reverses the assignment. **Fingerprint ordering contrast**: KDF_Call uses canonical (lexicographic) ordering — `fp_lo ‖ fp_hi` — because both parties must derive identical call keys from the same inputs. This is the opposite of the ratchet AAD construction (§5.3 Step 5), which uses direction-dependent ordering — `fp_sender ‖ fp_recipient` — because anti-reflection (Theorem 12) requires the AAD to differ between A→B and B→A directions. A modeler who builds a fingerprint-pair helper using canonical ordering (correct for KDF_Call) and reuses it for ratchet AAD would miss anti-reflection. Conversely, using direction-dependent ordering for KDF_Call would cause the two parties to derive different call keys.

Intra-Call Rekeying:

Periodically: (key_a', key_b', ck_call') ← KDF_CallChain(ck_call). The old ck_call and call keys are zeroized. Role assignment is preserved. The call chain is bounded at MAX_CALL_ADVANCE (2^{24}) advances: exactly MAX_CALL_ADVANCE advances succeed (step_count increments from 0 to 2^{24}), and the (MAX_CALL_ADVANCE + 1)-th advance fails with ChainExhausted (the guard `step_count ≥ MAX_CALL_ADVANCE` is checked before increment). All call key material is zeroized on exhaustion. For formal models, call chain exhaustion is a terminal state — no further intra-call rekeying is possible.

The ratchet state Σ is not modified by call key derivation — rk is read but not advanced. The ratchet continues operating independently for text messages during the call.

Call state: Each active call is represented by a state object Ψ_call = (key_a, key_b, ck_call, step_count), where step_count ∈ {0, ..., MAX_CALL_ADVANCE} tracks the number of intra-call rekeying advances. Ψ_call is created at Step 4 (KDF_Call) with step_count = 0, and destroyed when the call ends or step_count reaches MAX_CALL_ADVANCE (chain exhaustion — terminal state). Multiple Ψ_call instances may exist concurrently within a session, indexed by call_id (see Concurrent calls below). Ψ_call is independent of the ratchet state Σ: `Corrupt(RatchetState, P, t)` does not reveal any Ψ_call, and `Corrupt(CallKeys, P, call_id)` reveals only the Ψ_call for the specified call_id without affecting Σ or other active calls. A Tamarin model should represent each Ψ_call as a separate linear fact, consumed and re-created on each KDF_CallChain advance, and consumed without replacement on chain exhaustion or call termination. **Exhaustion semantics**: The protocol has two bounded resources with distinct exhaustion behavior. Call chain exhaustion (MAX_CALL_ADVANCE = 2^{24}) is terminal for the call instance — Ψ_call is destroyed — but the session remains alive (rk ≠ 0^{32}, new calls can be initiated). Receive-side counter exhaustion (recv_seen bounded at MAX_RECV_SEEN = 2^{16}, §5.1) is recoverable per epoch — recv_seen resets on KEM ratchet step (§5.4 Step 3c). Send-side counter exhaustion (s = $2^{32}-1$) is terminal for the send direction (§5.3 Step 1 caveat). A formal model that unifies these under a single "ChainExhausted" terminal state must distinguish per-call termination (Ψ_call destroyed, Σ unaffected), per-epoch recovery (recv_seen reset), and per-direction termination (send permanently blocked).

Epoch synchronization: Both parties must invoke KDF_Call with the same rk value — i.e., at the same ratchet epoch. The call offer (call_id, pk_eph) and response (c_eph) are transmitted as ratchet-encrypted signaling messages (Steps 1-2). If additional ratchet messages interleave between the call offer and key derivation — triggering a KEM ratchet step that advances rk on one side — the two parties derive call keys from different root keys, and the call fails (AEAD decryption of media frames produces ⊥). This is a correctness constraint, not a security violation: no key material is leaked, but the call is unusable. A formal model must order the `DeriveCallKeys` event between the same pair of KEM ratchet steps for both parties; allowing arbitrary interleaving of ratchet messages and call derivation produces false counterexamples to Theorem 8.

Concurrent calls: Multiple calls may be initiated within the same ratchet epoch, sharing the same rk as HKDF salt. Independence of their derived keys rests on call_id uniqueness: call_id is a 128-bit uniformly random value (Step 1), so two concurrent calls collide with probability ~2^{-64} at the birthday bound — negligible for any practical number of concurrent calls. When $\text{call\_id}_1 \neq \text{call\_id}_2$, the IKM inputs to KDF_Call differ ($\text{ss\_eph}_1 \| \text{call\_id}_1$ vs $\text{ss\_eph}_2 \| \text{call\_id}_2$), producing independent HKDF outputs by the PRF guarantee of HKDF-Expand on distinct inputs under the same extracted PRK. Consequently, `Corrupt(CallKeys, P, call_id₁)` does not reveal keys for a concurrent call with $\text{call\_id}_2 \neq \text{call\_id}_1$. A formal model supporting concurrent calls should include a freshness axiom: call_id values are unique per session with overwhelming probability (128-bit birthday bound).

```
Initiator                                        Peer
  |                                                                 |
  |  call_id ← random 128-bit                                       |
  |  (pk_eph, sk_eph) ← XWing.KeyGen()                              |
  |                                                                 |
  | ------------ [ratchet-encrypted] (call_id, pk_eph) ------------>|
  |                                                                 |
  |         (c_eph, ss_eph) ← Encaps(pk_eph)                        |
  |                                                                 |
  | <-------- [ratchet-encrypted] c_eph --------------------------- |
  |                                                                 |
  |  ss_eph ← Decaps(sk_eph, c_eph)                                 |
  |  zeroize sk_eph                                                 |
  |                                                                 |
  |  Both: (key_a, key_b, ck_call) ← KDF_Call(rk, ss_eph, call_id, fp_lo, fp_hi) |
  |  Both: zeroize ss_eph                                           |
  |  Role: lower fp → key_a=send, key_b=recv                        |
  |                                                                 |
  |  ================ E2E encrypted media stream ==================|
```

6. LO-Auth

Server → Client: c, where (c, ss) ← XWing.Encaps(pk_IK_client[XWing]) — where pk_IK_client[XWing] denotes the X-Wing component (first 1216 bytes) of the client's composite identity public key. Server retains token = MAC(key=ss, data="lo-auth-v1") locally; ss is zeroized immediately after token derivation.

Client → Server: proof = MAC(key=XWing.Decaps(sk_IK_client[XWing], c), data="lo-auth-v1") — where sk_IK_client[XWing] denotes the X-Wing component (first 2432 bytes) of the client's composite secret key.

Server: accept iff proof = token. **The comparison must be performed in constant time; timing side-channels on this comparison allow an adversary to forge authentication proofs.** ss was zeroized immediately after token was computed (above); token is zeroized after comparison.

```
Server                                          Client
  |                                                             |
  |  (c, ss) ← XWing.Encaps(pk_IK_client[XWing])                |
  |  token = MAC(key=ss, data="lo-auth-v1")                      |
  |  zeroize ss                                                 |
  |                                                             |
  | ------------------ c (challenge) ----------------------> |
  |                                                             |
  |         ss ← XWing.Decaps(sk_IK_client[XWing], c)           |
  |         proof = MAC(key=ss, data="lo-auth-v1")               |
  |         zeroize ss                                          |
  |                                                             |
  | <----------------------- proof ------------------------ |
  |                                                             |
  |  accept iff ct_eq(proof, token)                             |
  |  zeroize token                                              |
```

7. Security Properties

7.1 LO-KEX

Session key secrecy: The session key (rk, ek) is computationally indistinguishable from uniform to any PPT adversary that has not corrupted all keys necessary to reconstruct ikm: sk_IK_B (specifically its X-Wing component) and sk_SPK_B[id_SPK] and, when OPK was used, sk_OPK_B[id_OPK]. Corrupting any strict subset of these keys leaves at least one shared secret component (ss_IK, ss_SPK, or ss_OPK) underivable, and the HKDF output remains computationally indistinguishable from uniform (by the extraction property of HKDF, §8.4 — each X-Wing shared secret contributes 256 bits of min-entropy when its component KEM is unbroken, which exceeds the extractor's security parameter; a single uncompromised component therefore provides sufficient min-entropy for extraction regardless of the others). Additionally, the adversary must not have issued Corrupt(RNG, A, t) at the session establishment epoch; RNG compromise at encapsulation time allows reconstruction of all ephemeral shared secrets from the public transcript without corrupting any long-term key. The condition is asymmetric: only A's RNG is constrained because A performs all three encapsulations (§4.3 Steps 1-3). B's LO-KEX operations are deterministic decapsulations; B's RNG is not consumed until the first ratchet step (§5.2).

Multi-key forward secrecy: Corruption of sk_IK_B after a session is established, where sk_SPK_B[id_SPK] has been deleted, does not reveal the session key. Corruption of sk_SPK_B alone is likewise insufficient (ss_IK is also required). Corruption of sk_SPK_B[id_SPK] after sk_IK_B has been deleted (e.g., on identity key rotation) is likewise insufficient to reconstruct the session key. These claims are predicated on the session key having been fresh at establishment (see Session key secrecy

above); under Corrupt(RNG, A, t) at the establishment epoch, the session key was trivially derivable from the start and forward secrecy is vacuously satisfied.

Recipient authentication: Binding to B's identity key is twofold. First, *implicit*: only the holder of sk_IK_B can decapsulate c_IK and thereby derive ss_IK. Without ss_IK, the session key (HKDF output) remains computationally indistinguishable from uniform to the adversary. Second, *explicit*: fp_IK_B = H(pk_IK_B) is embedded in SI (§4.3 Step 4) and therefore in Encode(SI), which is signed by Alice as σ_SI (§4.3 Step 5). The signed payload directly names B as the intended recipient, independently of KEM decapsulability — this is the basis for Theorem 2(a)'s explicit branch and simplifies formal verification (a Tamarin model can derive recipient binding from σ_SI without a KEM decapsulation lemma). Additionally, the SPK signature verified in §4.2 (`HybridSig.Verify(pk_IK_B, "lo-spk-sig-v1" ‖ pk_SPK_B, σ_SPK)`) prevents an adversary from substituting a rogue pk_SPK_B without sk_IK_B's signing component — authentication is enforced at both the bundle layer (signature check) and the session layer (KEM binding to c_IK). Note: per §3, sk_IK_B contains a dedicated Ed25519 signing key; Corrupt(IK, B) therefore yields both decapsulation and signing capability.

Initiator authentication: Alice signs the encoded SessionInit under sk_IK_A (§4.3 Step 5): σ_SI = HybridSig.Sign(sk_IK_A, "lo-kex-init-sig-v1" ‖ Encode(SI)). Bob verifies σ_SI against pk_IK_A before decapsulating (§4.4 Step 2). This is explicit proof-of-possession: any adversary lacking sk_IK_A cannot produce a valid σ_SI with non-negligible probability (HybridSig EUF-CMA). Both identity keys are additionally committed into the HKDF info field (§4.3 Step 3) — any IK substitution that evades signature verification also fails at first-message decryption. Since fp_IK_B is included in SI (§4.3 Step 4) and therefore in Encode(SI), recipient binding is directly derivable from σ_SI: the signed payload explicitly names the intended recipient, without requiring analysis of the KEM ciphertexts' implicit binding. Note: the AEAD AAD (§4.3 Step 6) additionally binds both fingerprints; the full-key binding is in the HKDF info field.

TOFU caveat: The signature proves Alice holds sk_IK_A, but does not prove that pk_IK_A belongs to the human "Alice". On first contact (no shared community, no prior reference), Bob cannot verify this binding. An adversary controlling the channel could substitute a different IK pair (pk_IK_X, sk_IK_X), forge a valid σ_SI under sk_IK_X, and impersonate Alice to Bob. This is trust-on-first-use (TOFU), identical to Signal, SSH, and all systems without centralized PKI. It is inherent, not a bug.

Session-init header integrity: Any modification to SI fields after the first message is encrypted invalidates the AEAD tag (aad_0 binds Encode(SI) in full).

KCI resistance: Corruption of sk_IK_A does not enable impersonation of B to A. Impersonating B requires sk_SPK_B or sk_OPK_B, which are independent of sk_IK_A. Note: Corrupt(IK, A) does enable an adversary to forge A's own pre-key bundles (the signing component of sk_IK_A can sign a malicious SPK, per §3); it does not enable impersonation of B to A. Even combined with Corrupt(RNG, A, t), impersonation of B to A still requires forging a bundle signed by sk_IK_B's signing component — the combined corruption does not provide this. Corruption of A's own pre-keys (sk_SPK_A, sk_OPK_A) is likewise orthogonal to B-to-A impersonation.

7.2 LO-Ratchet

Message confidentiality: Each message is encrypted under a distinct message key mk derived from a unique (epoch_key, counter) pair. IND-CPA + INT-CTXT security of XChaCha20-Poly1305 under independent single-use message keys mk.

Message authentication: Each ratchet message is authenticated via XChaCha20-Poly1305 (INT-CTXT property). The AEAD tag covers both the plaintext and the AAD — `"lo-dm-v1"` , sender and recipient fingerprints, and the encoded header `(pk_s, c_ratchet, n, pn)` . An adversary without the message key mk cannot forge or modify a message or its header without detection.

Anti-reflection: The AAD construction uses direction-dependent fingerprint ordering: `AAD = "lo-dm-v1" ‖ fp_sender ‖ fp_recipient ‖ Encode(H)` (§5.3 Step 5). For a message A→B, the AAD contains `fp_A ‖ fp_B` ; for B→A, it contains `fp_B ‖ fp_A` . This means an adversary cannot reflect a message from A→B back as if it were B→A — the AEAD tag verification fails because the AAD byte sequences differ (fingerprint order is reversed). This is distinct from the replay resistance provided by recv_seen (which prevents replaying a message in the same direction). A Tamarin modeler who uses a canonical (e.g., sorted) fingerprint ordering in their AAD rule would miss this property and produce false reflection-attack counterexamples; the AAD rule must preserve the sender-first ordering.

Forward secrecy (per-epoch): Forward secrecy is provided at the epoch level, not the per-message level. Within an epoch, all message keys are derivable from the epoch key; compromising the epoch key reveals all messages (past and future) within that epoch. Between epochs, the KEM ratchet step replaces the epoch key via KDF_Root with a fresh KEM shared secret; the old epoch key cannot be recovered from the new one (HKDF one-wayness). Note: on the receive side, prev_ek_r (§5.1) retains the immediately prior epoch's receive key for one additional epoch to handle late-arriving messages — forward secrecy for a receive epoch therefore requires two KEM ratchet steps (see Theorem 4 for the precise statement). This is an intentional design decision — per-message forward secrecy (via sequential chain advancement) protects against an adversary who compromises the chain key at a specific message position, but this threat is dominated by the fact that the epoch key, root key, and ratchet secret key share the same memory region. See §6.13 of the implementation specification for the full rationale.

Break-in recovery (PCS): After state compromise at time t, security is restored once at least one KEM ratchet step after t executes with uncompromised encapsulator randomness and the decapsulator's send ratchet key `sk_s` used at the recovery step was generated in a prior ratchet step whose state was uncompromised — equivalently, no `Corrupt(RatchetState)` covers the recovery step or any preceding step between the compromise point and the recovery, and no `Corrupt(RNG, decapsulator, t_keygen)` was issued at the decapsulator's key generation epoch for the ratchet key pair used in the recovery step (see §8.3 conditions F1-F4 for the complete formal predicate). The KEM ratchet is **single-sided**: only the encapsulator contributes fresh randomness per step; the decapsulator's contribution is the existing ratchet public key from a prior step. This is weaker than a DH ratchet under encapsulator RNG compromise, but stronger against a quantum adversary capable of solving discrete logarithm problems from the transcript — the primary motivation for KEM-based ratcheting. This trade-off is a known structural property of all KEM-based ratchets (cf. [ACD19] for the DH ratchet baseline; [HKS+22] for KEM-ratchet construction; [BFG+20] §4 for the specific security loss characterization of single-sided KEM freshness vs bidirectional DH freshness).

Design note (bidirectional KEM ratchet) : A bidirectional variant — where both parties contribute a KEM shared secret per ratchet step — would provide two-sided freshness analogous to a DH ratchet, restoring PCS even under simultaneous encapsulator RNG compromise. This was evaluated and rejected for LO-Ratchet v1: each message would carry both a KEM ciphertext (1120 bytes) and a fresh X-Wing public key (1216 bytes) in *both* directions, roughly doubling per-message overhead, and recovery would still require a full round trip. The single-sided design recovers PCS within one round trip under the realistic threat model (at most one party's RNG is compromised at a time) while keeping message overhead minimal. A future protocol version could offer bidirectional KEM as an opt-in mode for high-assurance channels where bandwidth is not constrained.

Message header integrity: The full ratchet header (pk_s, c_ratchet, n, pn) is bound into the AEAD AAD. Substituting a different pk_s or injecting a false c_ratchet is detected at decryption.

Out-of-order delivery: Counter-mode key derivation allows any message within an epoch (or the previous epoch, via the retained prev_ek_r) to be decrypted in O(1) regardless of arrival order. The recv_seen set (bounded at MAX_RECV_SEEN = 65536) tracks which counters have been successfully decrypted for duplicate detection. No skip cache or sequential chain advancement is needed.

Nonce uniqueness: Each (mk, n) pair is used at most once. mk = KDF_MsgKey(ek, counter) is unique for each (epoch_key, counter) pair; the 24-byte AEAD nonce is `0^20 ‖ BE32(counter)` (per §2.4). The ChainExhausted guard at $s = 2^{32}-1$ prevents counter rollover. On the initial epoch derived from KDF_KEX, counter 0 is retired: Alice initialises s = 1; Bob initialises r = 1. The session-init first message consumed counter 0 with a uniformly random nonce n_0 (not counter-derived) and a message key mk_0 = KDF_MsgKey(ek, 0); this ensures the session-init nonce domain (random) and the initial-epoch ratchet nonce domain (counter ≥ 1) are disjoint. In the computational setting, the nonce formats also provide structural separation: counter-derived nonces have the form 0^{20} ‖ BE32(counter) — 20 leading zero bytes followed by 4 counter bytes. A uniformly random 24-byte nonce matches this structure with probability 2^{-160}, which is negligible. Since the session-init first message and subsequent ratchet messages share the same epoch key (ek from KDF_KEX), they constitute a single AEAD instance — the IND-CPA/INT-CTXT game is played over one key, and nonce uniqueness must hold across both the random first-message nonce and subsequent counter nonces. The nonce format collision probability (2^{-160}) establishes this. On all subsequent epochs (one per KEM ratchet step), messages start at counter 0 under a fresh epoch key derived from a unique KEM shared secret via KDF_Root; nonce uniqueness rests on epoch-key uniqueness, not counter retirement.

Lemma (Nonce Uniqueness): No two AEAD invocations within a session use the same (key, nonce) pair. This property rests on four independent mechanisms, each necessary:

1. **Fork prevention** (§4.6): At most one live copy of Σ exists per session. Without this, two copies could encrypt with the same (ek, counter) pair. Enforcement: single-device session ownership, `epoch` monotonicity check on deserialization.
2. **Anti-rollback** (§5.1): `min_epoch` integrity prevents storage-layer replay from rewinding the ratchet to a prior state, which would re-derive the same epoch key and restart the counter from its prior value. This is a non-cryptographic trust assumption (§8.4): `min_epoch` integrity must be maintained independently of the ratchet blob.
3. **ChainExhausted guards** (§5.3 Step 1, §5.4 Step 1): Prevent counter overflow at $s = 2^{32}-1$ and $n^* = 2^{32}-1$ respectively, ensuring the counter space is never exhausted and restarted within an epoch.
4. **Initial-epoch counter retirement** (§4.3 Step 6, §4.4 Step 8, §4.6): $s = 1$ for Alice, $r = 1$ for Bob after session establishment. Prevents collision between the random session-init nonce (used with mk_0 = KDF_MsgKey(ek, 0)) and counter-derived ratchet nonces (counter ≥ 1).

Theorem 3 depends on this lemma: without nonce uniqueness, the IND-CPA/INT-CTXT reductions are invalid (the AEAD security game requires each encryption query to use a fresh nonce).

7.3 LO-Call

Call key secrecy: The call keys (key_a, key_b) are computationally indistinguishable from uniform to any PPT adversary that has not obtained both rk and ss_eph (where ss_eph is zeroized after key derivation — see Forward secrecy below). rk is bound into the HKDF salt; ss_eph is bound into the IKM. Obtaining either alone is insufficient. `rk` is pseudorandom (output of KDF_Root or KDF_KEX — §2.3); the HKDF security reduction in §8.4 applies. Direct revelation via `Corrupt(CallKeys, P, call_id)` (§8.2) trivially violates secrecy for the current and all subsequent intra-call epochs; prior epochs remain protected by intra-call forward secrecy (see below).

Forward secrecy (ephemeral KEM): The ephemeral X-Wing keypair (pk_eph, sk_eph) is generated per call and zeroized after key derivation. Compromise of rk before or after the call does not reveal call content, provided ss_eph was not compromised — ss_eph is zeroized after key derivation and is no longer recoverable. Forward secrecy holds provided sk_eph and ss_eph are securely zeroized from memory after key derivation (as performed by the implementation). This is strictly stronger than deriving call keys from rk alone.

Defense-in-depth (post-quantum): rk in the HKDF salt carries the ratchet chain's accumulated post-quantum security. If the ephemeral KEM is broken by a quantum computer, rk still protects the call keys. If rk is compromised classically, the ephemeral KEM still protects.

Intra-call forward secrecy: KDF_CallChain advances the call chain one-way (HMAC-based). Compromise of a later call key does not reveal earlier media segments. The old chain key is zeroized after each advance.

No ratchet state mutation: Call key derivation reads rk but does not modify Σ. The ratchet continues operating independently; text messages sent during a call advance the ratchet as normal.

Signaling channel dependency: The call setup values (call_id, pk_eph, c_eph) are transmitted as ratchet-encrypted signaling messages (§5.7 Steps 1-2). The load-bearing property is **integrity** (INT-CTXT), not confidentiality (IND-CPA). If an adversary can modify c_eph in transit, they can substitute c_eph' = Encaps(pk_eph) with a known ss_eph, breaking call key secrecy. The ratchet's AEAD authentication prevents this. Confidentiality of the signaling channel is defense-in-depth: pk_eph is a KEM public key and c_eph is a KEM ciphertext — both are designed to be public. Call key secrecy rests on X-Wing IND-CCA2 for c_eph and on sk_eph being ephemeral/zeroized, not on the signaling channel being confidential. A modular proof of Theorems 8-10 building on Theorem 3 therefore depends on ratchet message authentication for the signaling messages, not on confidentiality — this halves the reduction chain.

7.4 LO-Auth

Key possession: An adversary without sk_IK_client (specifically its X-Wing secret key component) cannot produce a valid proof with non-negligible probability, by IND-CCA2 of X-Wing and PRF security of HMAC.

Replay resistance: Freshness of c is guaranteed by the randomness of XWing.Encaps. Each challenge produces a distinct shared secret with overwhelming probability; the server retains the derived token for a single comparison and zeroizes it, preventing acceptance of a replayed challenge. (This assumes the server's encapsulation uses a CSPRNG with sufficient entropy. Under `Corrupt(RNG, Server, t)` at the challenge epoch, the adversary can reconstruct the shared secret from the Encap randomness and replay resistance is not guaranteed; see §8.3 LO-Auth freshness predicate.)

Timing safety: The token comparison must be performed in constant time (§6); timing side-channels on this step allow adversarial token forgery via progressive refinement attacks.

Challenge timeout: The server SHOULD reject proofs received more than T seconds after the challenge was issued (recommended T = 30). This is an application-layer obligation, not enforced by the cryptographic protocol. A formal model should introduce a clock and reject proofs outside the validity window; without this bound, a passive adversary who later compromises sk_IK_client can replay a captured (c, proof) pair at any future time.

No channel binding: The proof `MAC(key=ss, data="lo-auth-v1")` is not bound to a session ID, connection context, or transport-layer identifier. It is effectively a bearer token — an adversary who intercepts a valid proof (e.g., via MitM on the transport before TLS establishment, or via application memory access) can present it on a different connection within the timeout window. Channel binding is delegated to the transport layer (QUIC with PQ TLS). Theorem 6 claims key possession proof, not channel-bound authentication; a formal model should not attribute channel binding to LO-Auth.

7.5 Considered and Ruled-Out Attacks

The following attack vectors were explicitly evaluated during protocol design. Each is documented here so that formal verifiers can confirm the ruling rather than rediscover the analysis.

A1. crypto_version substitution in relayed bundles: An adversary controlling the bundle relay could tamper with the `crypto_version` field in a pre-key bundle. Since `crypto_version` is intentionally excluded from the SPK signature (§4.1), the signature does not detect this modification. **Ruling**: The hard-fail version policy (§4.2 Step 3) rejects any unrecognized `crypto_version` unconditionally — there is no fallback negotiation, no "best supported" logic, and no silent downgrade. Tampering with `crypto_version` produces the same outcome as dropping the bundle entirely; the attacker gains no advantage over the Dolev-Yao baseline. **Coupling note**: This ruling is tightly coupled to the hard-fail policy. If a future protocol version introduced version negotiation or graceful fallback, `crypto_version` would need to be included in `σ_SPK` to maintain downgrade resistance.

A2. Bidirectional KEM ratchet for two-sided PCS: A variant where both parties contribute a KEM shared secret per ratchet step was evaluated (see §7.2 design note). This would provide PCS recovery even under simultaneous encapsulator RNG compromise. **Ruling**: Rejected for LO-Ratchet v1. Each message would carry both a KEM ciphertext (1120 bytes) and a fresh X-Wing public key (1216 bytes) in both directions, roughly doubling per-message overhead. Recovery still requires a full round trip. The single-sided design recovers PCS within one round trip under the realistic threat model (at most one party's RNG is compromised at a time). The trade-off is explicit: weaker under simultaneous bilateral RNG compromise, stronger on bandwidth.

A3. KCI via identity key compromise: Corruption of `sk_IK_A` might enable impersonation of B to A. **Ruling**: Impersonating B requires producing a valid bundle signed by `sk_IK_B` (specifically, `σ_SPK` must verify under `pk_IK_B`). `Corrupt(IK, A)` yields `sk_IK_A`, not `sk_IK_B`; the signing components are independent. Even combined with `Corrupt(RNG, A, t)`, impersonation of B to A still requires forging `σ_SPK` under `sk_IK_B`. See §7.1 KCI resistance.

A4. Session-init replay: An adversary replays a captured `(SI, σ_SI, n_o ‖ c_o)` to Bob. **Ruling**: The replay is a valid session init — `σ_SI` verifies, all KEM ciphertexts decapsulate correctly, and the first message decrypts. This is by design: the protocol cannot distinguish a replayed session init from a legitimate one without server-side nonce tracking (which is out of scope for E2E). The consequence is a duplicate session, not a security violation — Bob derives the same `(rk, ek)` as before, and subsequent ratchet messages under the duplicate session are indistinguishable from the original. The application layer is responsible for detecting duplicate sessions (e.g., via session identifiers or message-level deduplication).

A5. OPK reuse: If an OPK secret key is not deleted after first use, a second session init using the same OPK ciphertext would derive the same `ss_OPK`, weakening the forward secrecy guarantee. **Ruling**: §4.4 Step 6 mandates irreversible deletion of `sk_OPK` before the ratchet state is used for messaging. This is a normative protocol requirement, not merely an implementation recommendation. A Tamarin model should represent OPK deletion as a separate action fact (as noted in §4.4).

A6. Cross-protocol signature/MAC reuse: An adversary captures a signature `σ_SPK` and attempts to use it as `σ_SI`, or captures an HMAC token from LO-Auth and replays it as a chain KDF output. **Ruling**: Prevented by domain separation (Theorem 7). All signatures, HMAC tokens, HKDF derivations, and AEAD encryptions use disjoint labels/info strings. The label is always the first component of the signed/MACed message, so no adversarial relabeling can produce a valid transcript in a different component without breaking the underlying primitive's security.

A7. Nonce reuse via counter exhaustion: If `s` reaches `2³²−1` and wraps to 0, the same `(mk, nonce)` pair could be reused. **Ruling**: The `ChainExhausted` guard (§5.3 Step 1, §5.4 Step 1) fails the operation at `s = 2³²−1` (send) or `n* = 2³²−1` (receive), preventing counter rollover. The receive-side check executes before any epoch identification or KEM ratchet step, ensuring no state mutation occurs on a crafted counter value. This is a hard failure — the chain cannot produce further messages. In practice, 2^{32} messages on a single chain without a direction change is implausible (a KEM ratchet step resets the counter).

A9. Unknown Key Share (UKS) : Alice believes she shares a session with Bob, while Bob believes he shares it with Carol. **Ruling**: Prevented by dual fingerprint binding. Both fp_IK_A and fp_IK_B are bound into σ_SI (§4.3 Step 5 — the signed payload includes Encode(SI), which contains both fingerprints), into Encode(SI) itself (used as AEAD AAD in §4.3 Step 6), and into the HKDF info field (§4.3 Step 3 — both full composite public keys pk_IK_A and pk_IK_B are length-prefixed inputs). An adversary attempting to redirect A's session to Carol would need to substitute pk_IK_B with pk_IK_Carol in the info field (changing the HKDF output and thus the session key) or in SI (invalidating σ_SI under EUF-CMA). The triple binding (signature, AAD, HKDF info) makes UKS infeasible under EUF-CMA of HybridSig and IND-CCA2 of X-Wing. A Tamarin modeler should verify this as a standard AKE property alongside KCI (A3), replay (A4), and impersonation (Theorem 2).

A8. Duplicate detection memory exhaustion: An adversary sends messages with many distinct counter values to grow the recv_seen set. **Ruling**: The recv_seen set is bounded at MAX_RECV_SEEN (65536) entries per epoch (§5.1 invariant g). The set stores only 4-byte counters (not 32-byte keys as in a skip cache), so the maximum memory footprint is ~256 KB per epoch. The set resets on each KEM ratchet step. In practice, epochs are short (a KEM ratchet step occurs on each direction change), so the set rarely approaches the cap.

---

8. Adversary Model

8.1 Channel

The communication channel is fully adversarially controlled (Dolev-Yao). The adversary may read, **drop**, delay, reorder, modify, and inject messages. The pre-key bundle relay is assumed adversarially controlled; the bundle signature is the sole protection against relay-substituted pre-keys.

Header observability: The ratchet header H = (pk_s, c_ratchet, n, pn) is transmitted in cleartext — it is bound into the AEAD AAD (§5.3 Step 5) but not encrypted. A passive adversary observes: pk_s (reveals epoch transitions — a new public key indicates a new epoch), c_ratchet (reveals when a KEM ratchet step occurred — present vs absent), n (message counter within the epoch), and pn (number of messages sent in the previous epoch). For Tamarin/ProVerif: every Send rule must emit an `Out(H)` fact so the adversary can pattern-match on header fields. This is standard for Double Ratchet variants (Signal has the same property).

Channel 2 scope: The Dolev-Yao adversary in this model can observe all Channel 2 metadata — connection graph, epoch structure, message sizes, and timing — as a direct consequence of controlling the channel and having access to all cleartext headers and ciphertext lengths. This is intentional and complete: the model grants the adversary maximum observable information at the network layer. Theorems 1–13 prove Channel 1 properties (content confidentiality, authentication, forward secrecy, integrity) and make no claims about Channel 2. A formal model that reduces adversarial advantage over metadata (e.g., by hiding header fields or equalizing ciphertext lengths) models a strictly stronger channel than the one defined here and produces results that do not apply to this protocol without additional transport-layer assumptions. The Channel 2 surface is cataloged in §8.5 and in Crypto.md §1.5.

Parties and sessions: Parties are identified by their identity key pair (sk_IK, pk_IK). A **session** s = (A, B, role, Σ) is produced by a LO-KEX exchange, where A is the initiator, B is the responder, role ∈ {initiator, responder} identifies the local party's role, and Σ is the ratchet state. Two sessions **match** if they share the same (pk_IK_A, pk_IK_B) pair, their initial root keys are identical, and their roles are complementary (one initiator, one responder). call_id identifies a distinct LO-Call invocation within a session. Each party may participate in multiple concurrent sessions. The corruption queries in §8.2 use P to denote a party, id for a pre-key identifier, call_id for a call, and t for a time point. The terms "time point" and "epoch" are used interchangeably in §8; both denote a discrete atomic event in the protocol execution trace.
RNG granularity: Each protocol step that consumes randomness (§4.3 Session Initiation, §5.2 KEM Ratchet Step, §5.7 Steps 1-2 Call Setup, §6 LO-Auth Challenge) is treated as a single atomic time point for Corrupt(RNG) purposes, even when the step internally performs multiple randomness-consuming operations (e.g., §4.3 performs KeyGen + three Encaps + nonce generation). Corrupt(RNG, P, t) compromises all randomness consumed within the protocol step at time point t. A Tamarin model should encode each such step as a single rule consuming one Fr(~r) fact; the freshness predicates in §8.3 reference protocol steps, not individual cryptographic operations.

8.2 Corruption Queries

Corruption is adaptive: the adversary may issue Corrupt queries at any point during the protocol execution, based on the transcript observed so far. This is the standard model for AKE security (cf. [ACD19], [CGCD+20]). Computational reductions under adaptive corruption may lose tightness (the simulator must guess the target session for challenge embedding); the theorems in §8.6 are stated asymptotically and do not claim tight reductions.

A complete security model should support the following:

Query	Effect
Corrupt(IK, P)	Reveals sk_IK_P
Corrupt(SPK, P, id)	Reveals sk_SPK_P[id]

Query	Effect
Corrupt(OPK, P, id)	Reveals sk_OPK_P[id]
Corrupt(RatchetState, P, t)	Reveals Σ_P at time t
Corrupt(CallKeys, P, call_id)	Reveals the current call key state (key_a, key_b, ck_call) for call_id at the time of the query; keys from prior intra-call rekeying epochs are zeroized and not revealed
Corrupt(StreamKey, P, stream_id)	Reveals the 32-byte stream encryption key for stream_id. A stream_id is the (key, base_nonce) pair — two streams are distinct iff they differ in key or base_nonce. In the common single-key-per-stream case, base_nonce alone suffices as an identifier. Stream keys are caller-provided (not derived from protocol state), so this query is independent of Corrupt(RatchetState) — compromise of one does not imply the other
Corrupt(RNG, P, t)	Compromises all randomness consumed by P at time t — covers al `KeyGen`, `Encaps`, and nonce generation operations executing at that time point

The following table enumerates all randomness-consuming operations and their executing party, making the Corrupt(RNG) coverage explicit:

Step	Operation	Randomness consumer
§3 (bundle)	HybridSig.Sign(sk_IK_B, pk_SPK_B) → σ_SPK	Responder (bundle publication)
§4.3 Step 1	XWing.KeyGen() → (pk_EK, sk_EK)	Initiator
§4.3 Step 2	XWing.Encaps(pk_IK_B), Encaps(pk_SPK_B), Encaps(pk_OPK_B)	Initiator
§4.3 Step 5	HybridSig.Sign(sk_IK_A, Encode(SI)) → σ_SI	Initiator
§4.3 Step 6	n_0 ← uniform $\{0,1\}^{192}$	Initiator
§5.2 Step 1	XWing.KeyGen() → (pk_s', sk_s')	Sender
§5.2 Step 2	XWing.Encaps(pk_r)	Sender
§5.7 Step 1	XWing.KeyGen() → (pk_eph, sk_eph), call_id ← uniform $\{0,1\}^{128}$	Call initiator
§5.7 Step 2	XWing.Encaps(pk_eph)	Call peer
§6	XWing.Encaps(pk_IK_client[XWing])	Auth server
§15.1	base_nonce ← uniform $\{0,1\}^{192}$	Stream encryptor

The two HybridSig.Sign operations consume randomness via the ML-DSA-65 component (Ed25519 signing is deterministic per RFC 8032 and consumes no per-sign randomness; ML-DSA-65 uses hedged signing per FIPS 204 §5.2 and mixes fresh randomness into each signing operation for fault-injection resistance). However, Corrupt(IK, P) already yields the full signing key — an adversary who holds sk_IK can sign arbitrary messages without access to the signing randomness. In the security model, these signing operations add no adversarial advantage beyond what Corrupt(IK) provides, and no freshness predicate depends on signing randomness integrity. A Tamarin model should still consume a Fr fact in the SessionInit and BundlePublish rules (for completeness of the symbolic randomness model), but the signing randomness does not appear in any theorem's security loss.

This makes the asymmetry noted in §7.1 ("only A's RNG is constrained") immediately visible and extends it: in the ratchet, only the sender's RNG matters per step (§5.2); in calls, both parties consume randomness at different steps; in LO-Auth, only the server consumes randomness.

Zeroization semantics: Zeroization of a value v at time t removes v from the adversary's potential knowledge for all t' > t. Corrupt queries issued after zeroization do not yield the zeroized value — `Corrupt(RatchetState, P, t)` reveals only values present in Σ_P at time t; intermediate values (KEM shared secrets, old epoch keys, call setup ss_eph) that were zeroized before t are not recoverable. Several freshness predicates in §8.3 rely on this property (e.g., LO-Call freshness requires ss_eph zeroization; LO-Ratchet forward secrecy requires old epoch key zeroization after KEM ratchet steps). In Tamarin, zeroized values should be modeled as linear facts consumed at the zeroization point; in ProVerif, as values bound to phases that terminate at zeroization.

Storage encryption is scoped out of this formal model. Storage keys are application-layer at-rest protection, not protocol keys — they do not participate in any interactive exchange, are not derived from protocol state, and their compromise does not affect session key secrecy, authentication, or forward secrecy guarantees. Theorem 7 covers storage domain separation only in the label-disjointness sense: `"lo-storage-v1"` is distinct from all other labels in the §8.6 domain labels table, so no adversarial relabeling can repurpose a storage AEAD ciphertext as a ratchet message ciphertext or vice versa. The full storage AAD construction — `"lo-storage-v1" ‖ version ‖ flags ‖ BE16(|channel_id|) ‖ channel_id ‖ BE16(|segment_id|) ‖ segment_id` — is specified in the implementation (Crypto.md §storage, `storage::build_aad`) but is not formally modeled here; storage key confidentiality and storage AEAD security are out of scope.

Active use of corrupted keys: After `Corrupt(RatchetState, P, t)` or `Corrupt(IK, P)`, the adversary possesses the revealed secret keys and may use them to actively participate in the protocol as P — encrypting messages, performing KEM ratchet steps (KeyGen + Encaps), decapsulating incoming KEM ciphertexts, and signing session inits. This is implicit in the Dolev-Yao + corruption model (the adversary controls the channel and holds the keys), but is stated explicitly for modelers: a Tamarin model should include rules that allow the adversary to invoke Send and Receive operations on behalf of a corrupted party using the revealed state, not merely observe the state passively. In the standard eCK/CK terminology, this corresponds to the adversary having access to a `Send(s, m)` oracle on corrupted sessions. The freshness predicates in §8.3 are calibrated to exclude exactly those sessions where such active adversarial participation defeats the security guarantee.

`Corrupt(RatchetState, P, t)` reveals the full ratchet state Σ_P at time t; from this state the adversary can derive all message keys within the current epoch via `KDF_MsgKey(ek, counter)` for any counter value without further corruption queries. With counter-mode derivation, "covers the relevant epoch" in §8.3 F1 means the

adversary holds the epoch key for that epoch. Note: Σ includes the send ratchet keypair (pk_s, sk_s); the adversary therefore learns the KEM secret key that the peer will next encapsulate to. This is why F3 (§8.3) requires the decapsulator's state to be uncompromised at the recovery step — if sk_s is known, the adversary can compute the KEM shared secret from the public ciphertext, defeating recovery. **CCA oracle implication**: between `Corrupt(RatchetState, P, t)` and the next KEM ratchet step by the peer, the adversary holds sk_s and can inject messages with arbitrary c_ratchet values, triggering Decaps(sk_s, c_ratchet*) at §5.4 Step 3c and observing whether AEAD succeeds or fails. This constitutes a chosen-ciphertext oracle on the compromised KEM key. The proof of Theorem 5 (break-in recovery) therefore requires X-Wing IND-CCA2, not merely IND-CPA — the adversary can adaptively choose ciphertexts to decapsulate against the compromised key before the recovery step replaces it.

8.3 Freshness

A key material is **fresh** if no combination of corruption queries makes it trivially derivable.

LO-KEX: The session key is fresh if the adversary has not obtained all of {sk_IK_B (specifically its X-Wing component), sk_SPK_B[id_SPK] — obtainable only via Corrupt(SPK, B, id_SPK) issued while sk_SPK_B[id_SPK] is retained (during the grace period, §3); a Corrupt query after grace-period deletion yields no key material, and (when OPK was used) sk_OPK_B[id_OPK] — obtainable only via Corrupt(OPK, B, id_OPK) issued before OPK deletion (§4.4 Step 6; a Corrupt query after deletion yields no key material)}, has not issued Corrupt(RNG, A, t) at the session establishment epoch (consistent with §7.1), and has not issued Corrupt(RatchetState, P, t) at or after the establishment epoch for either party (which would directly reveal rk and ek). Corrupting any strict subset of the long-term keys leaves at least one shared secret component underivable.

LO-Ratchet: A message key is fresh if all of the following hold: **(F1)** no `Corrupt(RatchetState)` covers the relevant epoch; **(F2)** if `Corrupt(RatchetState, P, t_c)` was issued for any `t_c` before the target epoch, then at least one KEM ratchet step after `t_c` and before the target epoch — call it the **recovery step t_r** — must have executed with uncompromised encapsulator randomness (no `Corrupt(RNG, encapsulator, t)` for that step), establishing a fresh KEM shared secret that breaks the adversary's state continuity; absent any such prior corruption, F2 is vacuously satisfied; **(F3)** the decapsulator's `RatchetState` was uncompromised at `t_r` and all preceding steps between `t_c` and `t_r` (no `Corrupt(RatchetState)` covering those positions); **(F4)** the decapsulator's ratchet key pair (denoted `(pk_s, sk_s)` in §5.1) encapsulated to at `t_r` was generated without `Corrupt(RNG, decapsulator, t_keygen)` at its generation epoch (otherwise the adversary holds `sk_s` and can compute the KEM shared secret from the public ciphertext, defeating recovery). All four conditions are conjunctive. In the absence of any `Corrupt(RatchetState)` query before the target epoch, F2-F4 are vacuously satisfied and freshness reduces to F1 alone (no ratchet state has been compromised, so no recovery step exists and no recovery conditions need to be checked). **Syntactic nature**: F1-F4 are syntactic (query-based), not semantic (knowledge-based). After `Corrupt(RatchetState, P, t_c)`, the adversary can derive P's state at future time points without issuing new Corrupt queries — by tracking KEM ratchet steps where they hold the relevant sk_s. F3 is formally satisfied (no new Corrupt query was issued) even though the adversary may know the state through derivation. The security argument bridges this gap: at the recovery step t_r, the adversary's derived knowledge of rk is insufficient to compute ss' because sk_s at t_r is unknown (guaranteed by F4) — recovery relies on the IND-CCA2 reduction embedding its challenge at this step. A modeler who interprets F3 as "the adversary doesn't know the state" (semantic) rather than "no Corrupt query was issued" (syntactic) would produce an overly restrictive freshness predicate that excludes valid recovery scenarios. **Adversarial derivation bound (load-bearing sub-lemma for Theorem 5)**: After Corrupt(RatchetState, P, t_c), the adversary can extend their knowledge forward through ratchet steps where sk_s is known (i.e., sk_s was derived from state that was corrupt or from a KEM step they can invert). This chain terminates at the first step t* where sk_s was generated with uncompromised RNG — no Corrupt(RNG, decapsulator, t_keygen) was issued at t*'s keygen epoch (F4). The recovery step t_r is precisely such a t*: the adversary cannot compute ss' = Decaps(sk_s_t_r, c_ratchet) because sk_s_t_r is unknown, breaking the chain. This is the IND-CCA2 reduction's embedding point. A formal proof of Theorem 5 must include this bound as an explicit lemma — without it, the adversary's forward derivation chain might extend past t_r if the proof omits the F4 constraint, and a Tamarin formalization that models forward derivation symbolically must include a fact that is consumed when sk_s_t_r is generated (ensuring the derivation chain cannot traverse that step without a new Corrupt query). Incompatible reductions arise if reviewers treat F3 alone (no new Corrupt query) as sufficient without invoking the F4-termination argument — F3 syntactically prevents new queries but does not prevent forward derivation from prior state unless combined with F4.

LO-Call: (key_a, key_b) for call_id are fresh if the adversary has not both (a) obtained rk (via Corrupt(RatchetState) covering the derivation epoch, or by any combination of corruption queries that violates LO-KEX session key freshness (§8.3 LO-KEX predicate) and allows forward derivation of `rk` through all intermediate `KDF_Root` steps up to the derivation point) and (b) compromised `ss_eph` — via `Corrupt(RNG, encapsulator, t)` at the ephemeral encapsulation step (§5.7 Step 2) or via `Corrupt(RNG, initiator, t')` at the ephemeral KeyGen step (§5.7 Step 1), either of which yields the Encap randomness or `sk_eph` from which `ss_eph` is derivable. `Corrupt(CallKeys, P, call_id)` trivially violates freshness for the current and all subsequent intra-call epochs of the queried call; prior epochs remain fresh (§8.2 reveals only the current key state). **Composed-setting extension**: The above two conditions are sufficient for the standalone Theorem 8 game, which assumes the call setup messages (call_id, pk_eph, c_eph) are delivered without modification to the intended peer. In the primary deployment pattern — where these messages are transmitted as ratchet-encrypted signaling (§5.7 Steps 1-2) — ratchet message integrity (INT-CTXT, Theorem 3) is an additional implicit precondition: an adversary who can modify c_eph in transit can substitute a self-generated ciphertext with a known ss_eph, breaking call key secrecy without violating condition (b) above (no Corrupt(RNG) at the honest encapsulation step was needed) (see §7.3 signaling channel dependency note). A formal model of the composed system must include the ratchet message freshness predicate (§8.3 LO-Ratchet F1 applied at the signaling message epoch) as an additional condition for LO-Call key freshness; omitting it admits c_eph substitution as an in-model attack path. A standalone Theorem 8 proof — treating c_eph as honestly produced and delivered — does not require this condition; only the composed proof does.

LO-Auth: The authentication token is fresh if the adversary has not issued Corrupt(IK, client), has not issued Corrupt(RNG, Server, t) at the challenge epoch, and the server has not previously accepted a proof for the same challenge ciphertext c (single-use enforcement). The single-use condition is a server-side obligation: KEM encapsulation produces a fresh c with overwhelming probability (X-Wing randomness), but an active adversary who captures a valid (c, proof) pair can replay it unless the server rejects duplicate challenges. **Modeling note**: The server's challenge lifecycle should be modeled as a linear fact `AuthPending(server, client, token)` produced by the `AuthChallenge` rule and consumed by either: (a) `AuthVerify` (successful verification → produce `AuthAccepted(server, client)`), or (b) `AuthTimeout` (challenge expiry, recommended T=30s per §7.4 → consume without producing `AuthAccepted`). This parallels the OPK treatment (§4.4 Step 6), where single-use is enforced by consuming the `!OPK(id, sk)` linear fact. Without the linear fact structure, a model permits unbounded replay of valid (c, proof) pairs, which violates the single-use freshness condition.

Streaming AEAD: A stream's chunk confidentiality and integrity hold if the adversary has not issued `Corrupt(StreamKey, P, stream_id)` and has not issued `Corrupt(RNG, P, t)` at the base nonce generation epoch (where t is the stream encryptor initialization step — the §15.1 base_nonce generation during encryptor creation). Stream keys are caller-provided and independent of the protocol's ratchet state — `Corrupt(RatchetState)` does not reveal stream keys, and `Corrupt(StreamKey)` does not reveal ratchet state. **Temporal parameter**: The stream key freshness predicate has no temporal parameter — unlike the ratchet's per-epoch F1 condition — because the stream key is not rotated during a stream's lifetime. ¬Corrupt(StreamKey, P, stream_id) means the key was not revealed at any point during the game; a Corrupt(StreamKey) query at any point (including post-finalization) violates the predicate and removes all security guarantees retroactively, since the adversary can now forge against any previously-authenticated chunk. **Game formulation note**: The intended semantics is adversary class restriction (not game-abort): the advantage bound holds for all adversaries A satisfying the freshness predicate — Corrupt(StreamKey) defines the class boundary, not a game-aborting event within an execution. In adversary class restriction (formulation (a)), Corrupt is not modeled as a valid oracle query in the game; in game-abort formulation (b), a valid Corrupt oracle aborts the game and assigns advantage 0 to that execution. Both formulations yield equivalent security statements for a standalone fresh-key game. The distinction becomes load-bearing in a multi-challenge or composed game: under (b), an adversary who observes Q ciphertexts and then issues Corrupt scores advantage 0 for that execution, but the abort does not prevent the adversary from attempting Q forgeries in a non-aborting execution — the game-abort only prevents the advantage from being credited in the execution where Corrupt was called. A CryptoVerif modeler should encode the freshness condition as a session restriction (the Corrupt event must not fire, expressed as a game hypothesis that the Corrupt session event never occurs) rather than an abort-on-Corrupt rule, to avoid admitting hybrid adversaries that observe ciphertexts in one session and then issue Corrupt in another. The base nonce provides defense-in-depth: even if two streams share a key (violating the freshness assumption), different base nonces produce different per-chunk nonces, preventing nonce reuse. **Corrupt(RNG) role**: The base nonce is public (transmitted in the cleartext header), so in the single-stream case `Corrupt(RNG)` reveals nothing the adversary doesn't already have — the clause is vacuously satisfied. It becomes load-bearing in the multi-stream case: an adversary who controls the RNG can force identical base_nonce values across

streams sharing the same key, collapsing the $N^2/2^{193}$ birthday bound (Theorem 13, multi-stream note) to certainty. A modeler working on a single-stream-only model can drop this condition; a modeler working on §9.11(a) multi-stream nonce injectivity must retain it. **Syntactic nature under key reuse**: The freshness predicate is per-stream-id, where stream_id = (key, base_nonce) (§8.2). If multiple streams share the same key k with different base_nonces, `Corrupt(StreamKey, P, (k, bn₁))` reveals k — which transitively compromises all streams sharing k. However, no `Corrupt(StreamKey, P, (k, bn₂))` query was issued, so the freshness predicate for stream (k, bn_2) is syntactically satisfied despite the adversary holding the key. This parallels the F1-F4 syntactic-vs-semantic distinction in the LO-Ratchet freshness predicate above. A formal model must bridge this gap: either (a) model the stream key as a shared persistent fact so that one Corrupt query reveals the key for all streams using it, or (b) define stream_id = key alone, collapsing all same-key streams into one corruption target. Option (a) is more precise for the multi-stream birthday analysis. **Corrupt rule conclusion structure (Tamarin)**: The Corrupt(StreamKey, P, (k, bn)) rule should conclude `Out(k)` only — emitting the newly revealed secret. Emitting `Out(k, bn)` or the full stream_id tuple implies base_nonce is secret, which misrepresents the construction: bn is already in attacker knowledge from the public stream header (§15.1 Out(base_nonce) in the StreamEncInit rule). In the shared-key model (option (a), `!StreamKeyShared(k, [Alice, Bob])`), concluding `Out(k)` from the shared fact directly encodes the transitivity — one Corrupt query releases k to the attacker for all streams using it, without a separate cross-party lemma.

8.4 Assumptions

Operation fallibility in the abstract model: The following table classifies each abstract operation as total (always produces a valid output) or fallible (may produce a failure indicator). This determines which Tamarin/ProVerif rules need failure branches and which can be modeled as deterministic rewrite steps.

Operation	Abstract model	Failure mode
XWing.KeyGen	Total	—
XWing.Encaps	Total	—
XWing.Decaps	Total (implicit rejection)	Degenerate ss propagates to AEAD (§2.1)
HybridSig.Sign	Total	—
HybridSig.Verify	Fallible	Returns 0 (reject)
AEAD.Enc	Total	—
AEAD.Dec	Fallible	Returns $\perp$
KDF_Root	Total	—
KDF_KEX	Total	—
KDF_MsgKey	Total	—
KDF_Call / KDF_CallChain	Total	—
HKDF-Extract / HKDF-Expand	Total	—
Encode (§2.5)	Total	—

Only HybridSig.Verify and AEAD.Dec produce observable failure events. All KEM, KDF, and signing operations are total functions in the abstract model — implementation-level error paths (allocation failure, entropy exhaustion) are outside scope (§8.5). XWing.Decaps is total because ML-KEM implicit rejection ensures a deterministic output for every (sk, c) pair; the output may be "wrong" (decapsulation of a malformed ciphertext), but this is indistinguishable from a valid shared secret until AEAD detection.

- X-Wing Decaps is a deterministic function of (sk, c). This is load-bearing for the rollback re-derivability argument in §5.4 (re-processing the same c_ratchet* after rollback re-derives the identical KEM shared secret). IND-CCA2 does not formally require deterministic decapsulation; a modeler using a generic IND-CCA2 KEM abstraction that permits randomized Decaps would invalidate the rollback argument. Both X25519 (scalar multiplication) and ML-KEM-768 Decaps are deterministic by construction.
- X-Wing is IND-CCA2 (from the hybrid theorem: at least one of X25519 or ML-KEM-768 is IND-CCA2, combined via the SHA3-256 combiner under the random oracle model). **Identity element note**: §2.1 specifies that if X25519 produces the identity element (all-zeros shared secret), the degenerate value is included in the combiner without abortion. Under the random oracle model, the combiner output is uniform regardless of input (including degenerate inputs), so IND-CCA2 holds trivially. However, the "at least one component IND-CCA2" hybrid argument has a subtlety: if the X25519 component degenerates ($ss_X = 0^{32}$), the hybrid argument must rely on ML-KEM-768 being IND-CCA2 to maintain indistinguishability — the degenerate X25519 component contributes no entropy. A computational proof that explicitly opens the combiner (rather than treating X-Wing as a black-box KEM) must handle this case in the hybrid step where X25519 is the challenge component.
- HybridSig is EUF-CMA (at least one of Ed25519 or ML-DSA-65 is EUF-CMA).
- HKDF-Extract is a randomness extractor via the dual-PRF property of HMAC-SHA3-256 [Krawczyk10]; HKDF-Expand is a PRF keyed by the extracted PRK. The combined scheme is alternatively modeled as a random oracle in some reductions. Note: Krawczyk's dual-PRF analysis targeted HMAC-SHA2 (Merkle-Damgard). The argument carries over to HMAC-SHA3-256 via the generic HMAC construction [Bel06], which establishes the PRF and dual-PRF properties of HMAC under compression function assumptions without requiring Merkle-Damgård structure specifically. The SHA3 sponge structure is less studied in the HMAC-specific literature than SHA2, but the [Bel06] generic construction argument applies to any compression function satisfying the required pseudorandomness assumptions. SHA3's native keyed mode (KMAC) would be more natural but is not used here due to HKDF's reliance on HMAC. The protocol uses HKDF in two distinct entropy regimes: KDF_KEX (§2.3) uses salt = 0^{32} with high-min-entropy IKM (concatenated KEM shared secrets — §7.1 justifies ≥256 bits from each uncompromised component), invoking the extraction lemma directly; KDF_Root (§2.3) uses salt = rk (a pseudorandom HKDF output) with IKM = ss (a single KEM shared secret), invoking the dual-PRF property to treat the pseudorandom salt as a valid extraction key. A computational proof must handle these two cases separately.
- HMAC-SHA3-256 is a PRF. The protocol requires multi-evaluation PRF security in two contexts: (i) **concurrent call derivations** — multiple KDF_Call invocations sharing the same extracted PRK (from the same rk) but distinct info strings (different call_id values) must produce jointly independent outputs (Theorem 10, concurrent calls §5.7); (ii) **call chain three-output independence** — KDF_CallChain derives key_a, key_b, and ck_call' from the same chain key via HMAC with

single-byte inputs 0x04, 0x05, 0x06 respectively; joint independence of these three outputs under the PRF guarantee is required for Theorem 9 (knowing key_a and key_b must not reveal ck_call', which would break intra-call forward secrecy). Both are standard consequences of the PRF definition (a PRF is indistinguishable from a random function under polynomially many evaluations), but the multi-query structure is specific to these protocol components and affects the concrete security loss.

- XChaCha20-Poly1305 achieves IND-CPA + INT-CTXT security (equivalently, IND-CCA security for the combined AEAD scheme) under uniformly random keys. **Key commitment is not assumed**: XChaCha20-Poly1305 is not a key-committing AEAD — there exist (k_1, n_1) and (k_2, n_2) that produce the same ciphertext for different plaintexts. The current design does not require key commitment because epoch identification (§5.4 Step 2) selects a single key before decryption — there is no trial decryption across multiple keys — and the AAD binds the full header (which determines the epoch). If a future protocol revision introduced any form of trial decryption (e.g., for header encryption), this assumption would need revisiting.
- SHA3-256 is modeled as a random oracle where needed for computational reductions (specifically required for the X-Wing combiner security reduction); collision-resistant in symbolic models.
- The application maintains `min_epoch` with integrity independent of the ratchet blob (§5.1). This is a non-cryptographic trust assumption: the epoch anti-rollback mechanism (§5.1) prevents nonce reuse via storage-layer replay only if an adversary with Write(StorageBlob) capability cannot also substitute `min_epoch`. Without this assumption, the adversary can substitute both the blob and the epoch floor, rewinding the ratchet to a prior state and causing (epoch_key, nonce) reuse. **Per-session requirement**: min_epoch must be stored and compared per session, identified by the (local_fp, remote_fp) pair. A single global min_epoch across sessions allows cross-session blob substitution: an attacker with Write(StorageBlob, session_B) could substitute session A's blob (which has a higher epoch than session B's min_epoch floor), passing the global epoch check while installing a foreign session's ratchet state. A Tamarin model's Deserialize rule must parameterize its monotonic counter by session identifier.
- **Ratchet blob integrity**: The formal model assumes that the Deserialize rule only processes honestly-serialized blobs (produced by a prior Serialize rule or revealed via Corrupt(RatchetState)). The implementation presupposes this: authenticated decryption of the storage layer must succeed before `from_bytes_with_min_epoch` is called — feeding unauthenticated or adversary-crafted byte strings may produce a structurally valid but semantically corrupted state, equivalent to Corrupt(RatchetState) without issuing the query. An adversary with Write(StorageBlob) and the ability to forge authenticated blobs bypasses all deserialization invariants (a)-(s). This is distinct from the min_epoch assumption above: min_epoch prevents replay of *honestly-produced* old blobs, while blob integrity prevents *adversary-crafted* novel blobs.

8.5 Out of Scope

- **Deniability**: No deniability properties are claimed. KEM ciphertexts are attributable.
- **Group messaging**: The protocol is strictly two-party.
- **Traffic analysis / metadata (Channel 2)**: This document's security theorems cover Channel 1 — the content and integrity of transmitted data. They make no claims about Channel 2 — the structural metadata of communication. A network-level adversary (Dolev-Yao) can observe the following from honest protocol traffic, and no theorem in this document bounds adversarial advantage over this information:
  - *Connection graph*: who initiates a session with whom (revealed by bundle fetch and `SessionInit`).
  - *Epoch structure*: when KEM ratchet steps occur (from `pk_s` changes in the cleartext header), how many messages are in each epoch (`n`, `pn` fields).
  - *Message sizes*: total wire message = cleartext ratchet header (1225 bytes when no KEM step, 2347 bytes with KEM step) + AEAD ciphertext (compressed plaintext + 16-byte Poly1305 tag). The cleartext header dominates on-wire size; the AEAD ciphertext approximates compressed plaintext length.
  - *Timing*: message cadence and epoch transition timing.
  - *Probe surface*: a responder that structurally rejects a `SessionInit` (wrong crypto version, malformed content) responds differently from one that never received it. An adversary can probe whether a party is online or running a specific crypto version by sending crafted session inits and observing the response. This is a known Channel 2 leak; probing resistance requires transport-layer measures outside this model. Applications requiring metadata privacy must add transport-layer measures (padding, cover traffic, onion routing, encrypted transport wrapping the ratchet output) above this library.
- **Endpoint security**: Application-layer compromise above the protocol boundary is not modeled.
- **Side-channel attacks**: Timing, power, and cache-based attacks are generally outside scope at the formal model level; implementation-level mitigations are a separate concern. Exception: the constant-time comparison required by §6 is a protocol-level normative requirement that must be satisfied by any conforming implementation.

8.6 Formal Verification Targets

The following theorem statements are intended as verification targets for Tamarin, ProVerif, or comparable tools. Each references the freshness predicates in §8.3 and the assumptions in §8.4. Each theorem specifies its security game to make the statement self-contained.

Theorem 1 (LO-KEX Session Key Secrecy): *Game*: Real-or-Random key indistinguishability — challenger runs LO-KEX honestly, adversary issues corruption queries per §8.2, then receives either the real session key or a uniformly random key and outputs a bit. For any sessions established between parties A and B, if the session key is fresh (§8.3 LO-KEX predicate — in particular, Corrupt(RNG, A, t) must not have been issued at the session establishment epoch), then no PPT adversary can distinguish the session key from a uniformly random value with non-negligible advantage. Reduces to: X-Wing IND-CCA2, HKDF extractor property (§8.4). **Compositional note (LO-Auth)**: In the composed setting where LO-Auth and LO-KEX share sk_IK[XWing], LO-Auth sessions provide the adversary with an interactive Decaps oracle on the identity key: choose c, learn MAC(Decaps(sk_IK[XWing], c), "lo-auth-v1"). The proof of Theorem 1 must account for these as additional CCA2 oracle queries on the same key. Under IND-CCA2 this is sound (the CCA2 game already provides a decapsulation oracle), but the security loss is additive in the number of LO-Auth sessions — each LO-Auth challenge constitutes one additional decapsulation query to the IND-CCA2 challenger.

Theorem 2 (LO-KEX Mutual Authentication): *Game*: Entity authentication — adversary controls the network (Dolev-Yao), issues corruption queries, and attempts to cause a party to accept a session with a peer who did not participate. The adversary wins if a party accepts and the matching session's partner is not corrupted. (a) *Recipient authentication*: If A completes LO-KEX with B's verified bundle and obtains session key k, and k is fresh, then B possesses sk_IK_B. Binding is twofold: (i) *implicit* — only the holder of sk_IK_B can decapsulate c_IK and derive ss_IK, so a fresh session key cannot be derived without sk_IK_B (reduces to: X-Wing IND-CCA2 and HybridSig EUF-CMA — the latter ensures an adversary cannot substitute a rogue pk_SPK_B without B's signing key, which would bypass the implicit KEM binding on the SPK component); (ii) *explicit* — fp_IK_B = H(pk_IK_B) is embedded in SI and covered by σ_SI (§4.3 Steps 4-5), so the signed payload directly names B as the intended recipient, independently of KEM decapsulability. Full key confirmation occurs only when B's first ratchet message decrypts successfully. Contrapositive: forging a valid session requires Corrupt(IK, B). (b) *Initiator authentication*: If B successfully verifies σ_SI against pk_IK_A (§4.4 Step 2), then A possesses sk_IK_A. This is explicit authentication — σ_SI is proof-of-possession verified before any KEM operations. Contrapositive: producing a valid σ_SI without sk_IK_A is infeasible. Reduces to: HybridSig EUF-CMA. (c) *Key confirmation*: *Game*: Key confirmation under asynchronous establishment — adversary controls the network (Dolev-Yao) and attempts to cause A to accept a session key that B did not derive. Full key confirmation — A knows that B holds the same session key (rk, ek) — occurs only when B's first ratchet message decrypts successfully under A's session key. LO-KEX is asynchronous: A completes session establishment (§4.3) without any message from B, so at KEX completion A has no assurance that B has received or processed the session init. Key confirmation is deferred to the ratchet layer. **Confirmation chain**: B starts with pending = $\top$ (§4.6), so B's first reply triggers a KEM ratchet step (§5.2): B generates (pk_s_B, sk_s_B), computes ss = Encaps(pk_r = A's pk_s), derives (rk', ek_s_B) = KDF_Root(rk, ss), and encrypts under mk = KDF_MsgKey(ek_s_B, 0). Successful AEAD decryption by A proves: B holds mk → B holds ek_s_B → B holds rk (the session-derived root key, since ek_s_B = KDF_Root(rk, ss).epoch) → B completed KEX with the same session key. The confirmation proves B holds rk, not the session-derived ek directly — B's first message uses a ratchet-advanced epoch key, not the KEX-derived one. A Tamarin modeler implementing the key confirmation lemma must structure the proof through this chain (rk → KDF_Root → ek_s_B → KDF_MsgKey → mk) rather than a direct ek-based argument. A Tamarin model should distinguish the "session established" event (A completes §4.3) from the "key confirmed" event (A successfully decrypts a message from B); Theorems 2(a)/(b) cover the former, and this property covers the latter. Reduces to: XChaCha20-Poly1305 INT-CTXT (a valid ciphertext under the derived key proves possession of that key), Theorem 1 (session key secrecy ensures the derived key is unique to the session).

(d) *Key confirmation (B→A direction)*: B obtains key confirmation of A at session reception (§4.4 Step 8): successful AEAD decryption of the first message proves A

holds mk_0 = KDF_MsgKey(ek, 0), therefore A holds ek, therefore A derived the same (rk, ek) from KDF_KEX. Unlike 2(c), this requires no round trip — B confirms A immediately upon processing the session init. The confirmation chain is direct: $mk_0 \rightarrow ek \rightarrow (rk, ek)$ from KDF_KEX. Reduces to: XChaCha20-Poly1305 INT-CTXT, Theorem 1. **Asymmetry**: B confirms A at session reception (one message); A confirms B only after B's first ratchet reply (round trip). A Tamarin model should have two distinct key confirmation events: `KeyConfirmed_B(sid)` at §4.4 Step 8 and `KeyConfirmed_A(sid)` at A's first successful Decrypt from B.

Both guarantees are conditional on B having a valid binding between pk_IK_A / pk_IK_B and the intended parties (see TOFU caveat, §7.1).

Theorem 3 (LO-Ratchet Message Secrecy): *Game*: Left-or-Right message indistinguishability — adversary submits two equal-length plaintexts (m_0, m_1), receives the encryption of m_b for random b ∈ {0,1}, issues corruption queries per §8.2, and outputs a bit. For any ratchet message m at chain position (epoch, n), if the message key is fresh (§8.3 F1-F4), then no PPT adversary can recover m. Reduces to: Nonce Uniqueness Lemma (§7.2 — which in turn depends on fork prevention §4.6, min_epoch integrity §8.4, ChainExhausted guards §5.3/§5.4, and counter retirement §4.6), XChaCha20-Poly1305 IND-CPA + INT-CTXT, HMAC-SHA3-256 PRF, HKDF extractor (KDF_Root), X-Wing IND-CCA2.

Theorem 4 (LO-Ratchet Forward Secrecy): *Game*: Post-compromise key indistinguishability — challenger runs the ratchet, allows Corrupt(RatchetState) after KEM ratchet steps, then tests whether the adversary can distinguish prior-epoch message keys from random. Forward secrecy is per-epoch with a **two-step delay** on the receive side due to prev_ek_r retention (§5.1). After a single KEM ratchet step replaces P's receive epoch key, the old ek_r moves into prev_ek_r, which remains part of the ratchet state Σ. `Corrupt(RatchetState, P, t)` after this step reveals prev_ek_r and allows the adversary to derive all message keys from the immediately prior receive epoch via `KDF_MsgKey(prev_ek_r, counter)`. Forward secrecy for receive epoch E therefore requires **two** subsequent KEM ratchet steps: the first moves E's key into prev_ek_r, the second overwrites prev_ek_r (the old value is zeroized via `ZeroizeOnDrop`). On the send side, forward secrecy applies after a single step (ek_s is overwritten directly with no retention). Within an epoch, all message keys are derivable from the epoch key (counter-mode derivation); per-message forward secrecy is not provided. For Tamarin: a `Corrupt(RatchetState)` rule must output both ek_r and prev_ek_r as attacker knowledge; a lemma testing one-step receive-side forward secrecy will produce a valid counterexample. Reduces to: HKDF one-wayness (KDF_Root), X-Wing IND-CCA2. **Sub-lemma structure for verification**: Theorem 4 should be split into two independent Tamarin lemmas with known-correct expected outcomes:

- *Lemma 4a (SendEpochFS)*: After one KEM ratchet step advancing past send epoch E, `Corrupt(RatchetState, P, t)` for any t after the step does not reveal send-epoch keys from E. Expected: **succeeds** (ek_s is overwritten directly, no retention).
- *Lemma 4b (RecvEpochFS)*: After one KEM ratchet step advancing past receive epoch E, `Corrupt(RatchetState, P, t)` does not reveal receive-epoch keys from E. Expected: **produces counterexample** (prev_ek_r retains E's key). After **two** KEM ratchet steps past E, the lemma succeeds (prev_ek_r overwritten). The one-step counterexample is a useful sanity check confirming the model correctly captures prev_ek_r retention.

Theorem 5 (LO-Ratchet Break-In Recovery): *Game*: Post-compromise security — challenger runs the ratchet, allows Corrupt(RatchetState, P, t_c) at time t_c, then tests whether message keys derived after a recovery step t_r are distinguishable from random. After Corrupt(RatchetState, P, t_c), if conditions F1-F4 (§8.3) hold for a recovery step t_r > t_c, then message keys derived after t_r are fresh. Recovery latency: one KEM ratchet step (best case: one direction change after compromise — the compromised party sends, the peer responds). Worst case requires two direction changes if the peer sends first after compromise — the peer's message encapsulates to the compromised ratchet public key (the adversary holds sk_s from the state corruption and can compute the KEM shared secret from the public ciphertext c_ratchet), so that step is non-recovering; recovery occurs when the compromised party next sends, triggering a fresh KEM ratchet step with a newly generated keypair unknown to the adversary. Reduces to: X-Wing IND-CCA2, HKDF extractor, HMAC-SHA3-256 PRF (epoch key → message key derivation post-recovery).

Theorem 6 (LO-Auth Key Possession): *Game*: Impersonation under active attack (Dolev-Yao channel) — adversary controls the channel between client and server, may modify or replay challenges, interleave LO-Auth sessions with LO-KEX sessions (obtaining additional Decaps oracle queries on sk_IK[XWing]), and relay challenges across sessions. The adversary then attempts to produce a valid proof for a fresh challenge without holding sk_IK. If auth_verify(expected_token, proof) returns true, the client possesses sk_IK (X-Wing component). The active-attack setting is strictly stronger than passive observation and matches the §8.1 adversary model. Reduces to: X-Wing IND-CCA2 (handles adaptive chosen-ciphertext queries including cross-protocol oracle access), HMAC-SHA3-256 PRF (which implies MAC unforgeability).

Theorem 7 (Domain Separation): *Game*: Cross-component transcript forgery — adversary obtains valid outputs from one protocol component and attempts to present them as valid in a different component. No signature, HMAC token, HKDF output, or AEAD ciphertext produced by one protocol component (LO-KEX, LO-Ratchet, LO-Call, LO-Auth, Storage, Streaming) is valid in another. Follows from disjoint labels/info strings and non-overlapping AAD prefixes:

Label	Type	Component
`"lo-auth-v1"`	HMAC data	LO-Auth
`"lo-kex-v1"`	HKDF info prefix	LO-KEX
`"lo-spk-sig-v1"`	Signature domain separator	LO-KEX (SPK signing)
`"lo-kex-init-sig-v1"`	Signature domain separator	LO-KEX (SessionInit signing)
`"lo-ratchet-v1"`	HKDF info	LO-Ratchet (root KDF)
`"lo-dm-v1"`	AEAD AAD prefix	LO-Ratchet (message encryption); also used by the session-init first message (§4.3 Step 6) — separation between session-init and ratchet messages relies on Encode injectivity (Encode(SI) ≠ Encode(H) structurally) and distinct message keys (counter 0 for first message, counter ≥ 1 for ratchet)
`"lo-call-v1"`	HKDF info	LO-Call

Label	Type	Component
`"lo-storage-v1"`	AEAD AAD prefix	Storage (community)
`"lo-dm-queue-v1"`	AEAD AAD prefix	Storage (DM queue)
`"lo-stream-v1"`	AEAD AAD prefix	Streaming AEAD (§15)
`0x01`	HMAC domain byte prefix	LO-Ratchet (KDF_MsgKey) — separation from LO-Call is by **key separation** (KDF_MsgKey uses epoch keys ≠ KDF_CallChain keys); byte disjointness from 0x04/0x05/0x06 is defense-in-depth only (see primary separation argument below)
`0x04` / `0x05` / `0x06`	HMAC data byte	LO-Call (call chain) — separation from LO-Ratchet is by **key separation** (KDF_CallChain uses call chain keys ≠ KDF_MsgKey keys); byte disjointness from 0x01 is defense-in-depth only (see primary separation argument below)

Sub-lemma (Encode disjointness): The `"lo-dm-v1"` entry shares an AAD prefix between session-init first messages and ratchet messages. Cross-type confusion is excluded by a length argument. Encode(SI) wire layout (from `encode_session_init`, Crypto.md §6.3): `BE16(|crypto_version|) ‖ crypto_version ‖ fp_IK_A ‖ fp_IK_B ‖ pk_EK ‖ BE16(1120) ‖ c_IK ‖ BE16(1120) ‖ c_SPK ‖ spk_id ‖ has_opk [‖ BE16(1120) ‖ c_OPK ‖ opk_id]`. Minimum (no OPK, crypto_version = "lo-crypto-v1"): 14 + 32 + 32 + 1216 + 1122 + 1122 + 4 + 1 = **3,543 bytes**; maximum (with OPK): 3,543 + 1,126 = **4,669 bytes**. Encode(H) has maximum length 2,347 bytes (1216-byte ratchet_pk + 1-byte presence flag + when present: 2-byte big-endian length prefix ‖ 1120-byte KEM ciphertext + two 4-byte counters; 1225 bytes without KEM ciphertext). The minimum gap is 3,543 − 2,347 = **1,196 bytes**, so the two encodings have disjoint length ranges and are distinguishable without a type tag. Combined with distinct message keys (counter 0 for first message, counter ≥ 1 for ratchet), this provides two independent separation mechanisms. Concrete sizes are from Crypto.md §6.3 (wire format).

The primary domain separation guarantee is key separation: KDF_MsgKey operates on epoch keys (derived from KDF_Root), while KDF_CallChain operates on call chain keys (derived from KDF_Call) — these are independent HMAC keys, so no input collision can produce a cross-component forgery regardless of data field values. The call chain bytes (0x04/0x05/0x06) are deliberately disjoint from the ratchet message key domain byte (0x01) as a secondary defense-in-depth measure that provides label-level separation even in the hypothetical case of key reuse. Note: the verification phrase labels (`"lo-verification-v1"` and `"lo-phrase-expand-v1"`) and the X-Wing combiner label (§2.1) are omitted — the former two (`"lo-verification-v1"` and `"lo-phrase-expand-v1"`) are outside the formal model — they appear only in the display-layer key comparison phrase (§9.4) and not in any key-derivation or authentication construction, so no security property in this document depends on their domain separation, and the latter is internal to the X-Wing primitive.

Theorem 8 (LO-Call Key Secrecy): *Game*: Real-or-Random key indistinguishability — challenger runs call setup honestly, adversary issues corruption queries, then receives either the real call keys or uniformly random keys and outputs a bit. For a call with call_id initiated within a session with root key rk, if (key_a, key_b) are fresh (§8.3 LO-Call predicate), then no PPT adversary can distinguish the call keys from uniformly random values. Reduces to: X-Wing IND-CCA2 (ephemeral encapsulation), HKDF extractor property.

Theorem 9 (LO-Call Intra-Call Forward Secrecy): *Game*: Post-compromise key indistinguishability on the call chain — challenger advances the call chain, allows Corrupt(CallKeys) at a later epoch, then tests whether the adversary can distinguish prior-epoch call keys from random. After a call chain advance (§5.7 Intra-Call Rekeying), compromise of the current call key state does not reveal call keys from prior epochs. Reduces to: HMAC-SHA3-256 PRF (one-wayness of call chain KDF).

Theorem 10 (Call/Ratchet Independence): *Game*: Cross-component key indistinguishability — adversary issues Corrupt(CallKeys) or Corrupt(RatchetState) for one component, then attempts to distinguish keys from the other component from random. Corrupt(CallKeys, P, call_id) does not violate LO-Ratchet message secrecy (Theorem 3), and Corrupt(RatchetState, P, t) at time t after call derivation does not retroactively reveal call keys derived before t, provided LO-Call freshness (§8.3) holds — in particular, the ephemeral encapsulation step (§5.7 Step 2) must use uncompromised RNG, ensuring ss_eph was unpredictable. The ephemeral keypair (pk_eph, sk_eph) and shared secret ss_eph are generated within call setup (§5.7 Steps 1-3) and zeroized after HKDF — they are not part of the ratchet state Σ. Therefore Corrupt(RatchetState) does not yield ss_eph; recovering it from the public transcript (c_eph) requires breaking X-Wing IND-CCA2. Reduces to: HKDF dual-PRF assumption, X-Wing IND-CCA2. **Dual-use salt note**: rk serves as the HKDF salt in both KDF_Root (§2.3, ratchet advancement) and KDF_Call (§2.3, call key derivation). If a call is initiated and a KEM ratchet step occurs in the same epoch, the same rk value is used as salt in both an HKDF invocation with info = `"lo-ratchet-v1"` and an HKDF invocation with info = `"lo-call-v1"`. Independence of the two outputs follows from a hybrid argument: under the dual-PRF assumption, HKDF-Extract(salt=rk, ikm) produces a pseudorandom PRK; HKDF-Expand then acts as a PRF keyed by PRK, and distinct info strings produce independent outputs by the PRF guarantee. The combined security loss is additive (one PRF advantage term per HKDF invocation sharing the same salt), not multiplicative.

Theorem 11 (Concurrent Call Key Independence): *Game*: Multi-instance Real-or-Random — challenger runs multiple call setups with distinct call_id values within the same session (same rk), adversary issues Corrupt(CallKeys, P, call_id_i) for a subset, then receives either the real call keys or uniformly random keys for the remaining calls and outputs a bit. For calls with $call\_id_1 \neq call\_id_2$ derived under the same rk, the call keys (key_a_1, key_b_1) and (key_a_2, key_b_2) are jointly independent and individually indistinguishable from uniform. `Corrupt(CallKeys, P, call_id₁)` does not violate key secrecy for $call\_id_2$. Reduces to: HKDF dual-PRF assumption (distinct IKM inputs $ss\_eph_1 \| call\_id_1$ vs $ss\_eph_2 \| call\_id_2$ produce independent PRKs), HKDF-Expand PRF (distinct info strings `"lo-call-v1" ‖ fp_lo ‖ fp_hi` are identical across calls but the PRK inputs differ). The security loss is additive in the number of concurrent calls. See §9.5 for the verification target.

Theorem 12 (Anti-Reflection): *Game*: Reflection attack — adversary captures a message from A→B and presents it to A as if it were from B→A within the same session. For any ratchet message encrypted by A with AAD = `"lo-dm-v1" ‖ fp_A ‖ fp_B ‖ Encode(H)`, decryption by A (as if receiving from B) reconstructs AAD = `"lo-dm-v1" ‖ fp_B ‖ fp_A ‖ Encode(H)`. Since `fp_A ‖ fp_B ≠ fp_B ‖ fp_A` (guaranteed by invariant (h): local_fp ≠ remote_fp, which ensures the two 32-byte fingerprints are not identical and therefore their concatenation order matters), the AEAD tag verification fails. This is distinct from recv_seen replay resistance (which prevents replaying a message in the same direction) — anti-reflection prevents cross-direction replay. **Freshness predicate**: Anti-reflection holds for any message whose epoch key is fresh per §8.3 F1 (no Corrupt(RatchetState) revealing ek at the time of encryption), under invariant (h) (local_fp ≠ remote_fp). Reduces to: XChaCha20-Poly1305 INT-CTXT, invariant (h).

Theorem 13 (Streaming AEAD Chunk Security): *Game*: Multi-chunk Real-or-Random, parameterized by Q_seq (maximum sequential encryption oracle queries) and Q_par (maximum random-access encryption oracle queries via `encrypt_chunk_at`), with Q = Q_seq + Q_par total encryption invocations — adversary adaptively submits (plaintext, is_last) pairs, where is_last determines tag_byte (§15.2) and thus both the chunk nonce and AAD (§15.4); for each pair, the challenger returns either the real AEAD ciphertext (b=0) or an encryption of a random byte string of the same post-compression length (b=1) — the bit b is drawn uniformly at random by the challenger once at game start and is fixed for the entire experiment (all oracle responses are consistently real or uniformly random under the same b). The game operates at the AEAD input layer, after compression — the adversary submits post-compression byte strings directly; compression is a preprocessing step outside the

game, and the b=1 challenger encrypts a uniformly random string of the same submitted length without applying compression. The adversary controls the chunk transport (Dolev-Yao) and has access to both sequential and random-access decryption oracles on the same stream, which it may interleave with encryption queries. The adversary issues `Corrupt(StreamKey)` adaptively and attempts to determine b (confidentiality) or produce a forgery (integrity). **ROR vs LoR**: Theorem 13 uses Real-or-Random (ROR) rather than Left-or-Right (LoR) as in Theorem 3. Both games are equivalent up to a factor of 2 in advantage. ROR is used here because the multi-chunk hybrid argument (Security loss note below) switches one chunk from real to random at each step — this is cleaner to state under ROR, where the random response is a uniformly random string of the same length, than under LoR, where the adversary's (m_0, m_1) pair would need to be re-specified at each hybrid step. A researcher composing Theorem 3 (ratchet message secrecy, LoR) with Theorem 13 (streaming IND-CPA, ROR) may convert between the two games with the standard equivalence: $Adv_{LoR}(A) \leq 2 \cdot Adv_{ROR}(A)$ and $Adv_{ROR}(A) \leq 2 \cdot Adv_{LoR}(A)$. This conversion applies to the IND-CPA component only — the INT-CTXT notion is a single-adversary forgery game with no challenge bit b, so it has no LoR/ROR variant and the factor-of-2 does not apply to the INT-CTXT security loss in a composed proof. **Game separation**: The confidentiality goal (determine b) and the decryption oracles are not simultaneously active in a single experiment — a combined game where b is live and the adversary can query decryption on its own encryption outputs would be trivially winnable (encrypt m → decrypt C → determine b). The combined description is for presentational unity; IND-CPA (Property 1) and INT-CTXT (Property 2) are instantiated as separate sub-games composed via [BN00]. See Oracle scope note below. **Oracle scope**: The decryption oracles are relevant to Properties 2-5 (integrity, ordering, truncation, isolation). Property 1 (IND-CPA) is a standalone confidentiality claim that does not use the decryption oracles — the upgrade to IND-CCA follows from the composition of Property 1 + Property 2 via the Bellare-Namprempre composition result [BN00] (IND-CPA + INT-CTXT ⇒ IND-CCA), applied per-chunk following the per-chunk reduction in §9.11 — the aggregate IND-CCA advantage is additive in the number of chunks. Properties 3-4 are stated with respect to the sequential oracle; the random-access oracle provides neither. **Oracle independence for Properties 3/4**: Co-exposing `decrypt_chunk_at` does not weaken Properties 3 or 4 for the sequential oracle. The INT-CTXT reductions for both properties already model full decryption oracle access in the forgery set; the adversary's use of `decrypt_chunk_at` provides no ordering or finalization power beyond what the INT-CTXT reduction already captures. The §15.3 state partition is the structural reason: `decrypt_chunk_at` reads only persistent state (key, base_nonce) and cannot advance `next_index` or set `finalized` — the linear state fields that govern Properties 3 and 4 are untouched by the random-access oracle. A Tamarin model may therefore safely omit `decrypt_chunk_at` when proving Properties 3 or 4 without loss of soundness; retaining it is conservative but not required. **Nonce-misuse resistance**: This construction is not nonce-misuse-resistant (NMR). base_nonce reuse under the same key is catastrophic for IND-CPA: two streams with (key, base_nonce_A = base_nonce_B) produce identical per-chunk nonces at equal (chunk_index, tag_byte) positions, so XOR of their ciphertexts directly reveals XOR of plaintexts. This is a consequence of using XChaCha20-Poly1305 in the nAE model — security requires nonce uniqueness, not nonce secrecy. A formal modeler should not attempt to verify NMR (or its stronger form, MRAE — Misuse-Resistant AE, [RS06]) as a Theorem 13 corollary. The construction operates in the nAE model [Rog04]: security requires nonce uniqueness, not nonce secrecy, and nonce misuse is outside the security game. Treat base_nonce reuse as a freshness violation (via the Corrupt(RNG) precondition in Property 5 or a direct key-freshness violation for same-key streams). **Proof structure**: [HRRV15]'s OAE2 proof does not apply here (see §9.11 proof-structure note) — the random-access interface explicitly breaks OAE2's sequential ciphertext-chaining dependency. The correct reduction goes directly from per-chunk XChaCha20-Poly1305 IND-CPA + INT-CTXT under injective nonces (§15.2) and AAD binding (§15.4), with each oracle query reducing independently to per-chunk AEAD queries. **Parallel encryption scope**: The game's encryption oracle is sequential — the adversary submits (plaintext, is_last) pairs and receives responses at implicit indices. The parallel encryption interface (`encrypt_chunk_at` , §15.3) is not modeled as a separate game oracle: IND-CPA for `encrypt_chunk_at` reduces to the same per-chunk XChaCha20-Poly1305 reduction as the sequential path (it reads only persistent state). A formal model that exposes both encryption interfaces must enforce the distinct-(index, tag_byte)-pair constraint as an **explicit game-aborting precondition on adversary queries**: if the adversary submits an `encrypt_chunk_at(i, t)` query at (i, t) already used by the sequential oracle (i.e., already in the sequential oracle's log at that index), or vice versa, **or if (i, t) was already submitted to `encrypt_chunk_at` in a prior query (regardless of plaintext)**, the game aborts. The intra-parallel case is a distinct gap from the cross-oracle case: two calls `encrypt_chunk_at(i, t, m₁)` and `encrypt_chunk_at(i, t, m₂)` with $m_1 \neq m_2$ produce ciphertexts under the same nonce, so $C_1 \oplus C_2 = m_1 \oplus m_2$ is known to the adversary; in the b=1 branch the responses r_1 and r_2 are independently uniformly random, so their XOR is unpredictable — the adversary distinguishes b with overwhelming probability without touching the sequential oracle. Framing this as a caller obligation rather than a game rule admits in-game nonce reuse — the combined-oracle game would then include nonce-collision transcripts, violating the invariant that the INT-CTXT reduction handles a single key with distinct nonces. In Tamarin, both abort preconditions are explicit premises in the EncryptChunkAt rule: `not (i, t) ∈ SeqOracleUsed ∧ not (i, t) ∈ ParOracleUsed` . Note that `encrypt_chunk_at` accepts the full u64 index domain (0..u64::MAX) — the sequential encryptor's 0..2^{64}-2 bound (§15.2) does not apply. **nAE oracle framing for CryptoVerif**: Within a stream (fixed key and base_nonce), the `encrypt_chunk_at` interface is a nonce-based AE (nAE) encryption oracle [Rog04] in which (chunk_index, tag_byte) serves as the per-call nonce (since chunk_nonce = base_nonce XOR mask(chunk_index, tag_byte) is a bijection on (chunk_index, tag_byte) for fixed base_nonce). The distinct-(chunk_index, tag_byte)-pair constraint is precisely the nonce-respecting adversary restriction from [Rog04] §4 applied to this nonce space — it is not merely a "caller safety obligation" but the formal boundary inside which nAE security holds. The nAE model makes no security claim for nonce-misusing adversaries (cf. the Nonce-misuse resistance note above for the cross-stream analog). In CryptoVerif specifically, this distinction is load-bearing: the nonce-respecting restriction must be encoded as an explicit game hypothesis (a `distinct` or `neq` assumption on oracle query nonces), not as a conditional check inside the oracle body. A CryptoVerif oracle that branches on nonce reuse (e.g., `if nonce ≠ prev then encrypt else ⊥`) still admits nonce-collision transcripts in the game's probability space and may produce an unsound bound — the game hypothesis approach structurally excludes them. A Tamarin or EasyCrypt model that encodes the constraint as a guard premise (not a session-level axiom) has the analogous risk. **Forgery exclusion**: A formal model that exposes `encrypt_chunk_at` as a co-oracle should include its outputs in the forgery exclusion set alongside sequential oracle outputs. If `encrypt_chunk_at` is not exposed as a co-oracle, its outputs fall outside the forgery exclusion set and are formally valid forgery targets from the game's perspective. Under the fresh-key precondition, however, the adversary cannot produce such outputs without either holding the stream key (Corrupt(StreamKey), violating freshness) or forging the underlying AEAD (excluded by XChaCha20-Poly1305 INT-CTXT). Property 2 therefore holds regardless. The phrase "outside the oracle's output set" is used here in the formal game-boundary sense — these are honest ciphertexts, not fabricated ones; the adversary simply has no oracle path to obtain them under the freshness assumption. **Forgery definition**: Let L = L_seq ∪ L_par denote the forgery exclusion set, where L_seq is the set of (chunk_index, tag_byte, ciphertext) triples output by the sequential encryption oracle, and L_par is the set of triples output by the `encrypt_chunk_at` oracle (∅ when `encrypt_chunk_at` is not co-exposed). A query that triggers a game abort (e.g., a repeated (index, tag_byte) pair submitted to `encrypt_chunk_at`) produces no oracle output and is not added to L_par — the game abort occurs before any ciphertext is returned, so the aborted query's (index, tag_byte, ciphertext) is not in L_par even though the adversary attempted the call. When `encrypt_chunk_at` is not co-exposed, L = L_seq. A forgery is a triple (chunk_index, tag_byte, ciphertext) ∉ L that either decryption oracle accepts. For the random-access oracle, chunk_index is supplied as an explicit input. For the sequential oracle, 'accepted at chunk_index' means the oracle's internal next_index equals chunk_index at the time of the query — the adversary must advance the oracle's state to position chunk_index before presenting (tag_byte, ciphertext). **Oracle equivalence for INT-CTXT accounting**: At any given (chunk_index, tag_byte), both decryption oracles use identical key, nonce (base_nonce XOR mask(chunk_index, tag_byte), §15.2), and AAD (§15.4, same chunk_index and tag_byte as inputs). A (chunk_index, tag_byte, ciphertext) triple that fools the random-access oracle at that index is simultaneously valid against the sequential oracle at the same position (and vice versa, modulo the sequential oracle having advanced to that position). The "either" in the forgery definition specifies the two access paths through which the adversary may present the triple — it does not introduce two distinct security thresholds or create a gap where a forgery succeeds in one oracle's accounting but not the other's. A formal model instantiating two separate Tamarin oracle accounts for the sequential and random-access decryptors should share the underlying AEAD verification rule (same key, nonce derivation, and AAD constructor) so that a forgery attempt at (chunk_index, tag_byte, ciphertext) resolves identically in both accounts. **Sequential oracle termination**: The sequential encryption oracle has two distinct terminal states. (a) **Finalization** (is_last=true submitted): the oracle transitions to a finalized state, rejects all further encryption queries (returning ⊥), and is_finalized()=true. (b) **Counter exhaustion** (ChainExhausted): the oracle rejects an attempted query at next_index = $2^{64} - 1$ before use (returning ⊥), and is_finalized()=false — the stream has been exhausted without a genuine final chunk. A formal model must include both as distinct terminal transition rules. The exhaustion terminal state (b) is the streaming-AEAD parallel to the ratchet's ChainExhausted guard (§5.3); the stream integrity corollary (Properties 3+4) is vacuously inapplicable in the exhaustion case (no finalization occurred), while Property 2's INT-CTXT game continues via both decryption oracles. An unconstrained oracle that admits queries at next_index = $2^{64} - 1$ — which the implementation rejects — produces spurious counterexample traces for Properties 3 and 4 in formal models, in the same way an unconstrained post-finalization decryptor would (see below). When `encrypt_chunk_at` is co-exposed, L_par may still grow after sequential counter exhaustion, for the same reason as after sequential finalization. **Encryption oracle finalization does not terminate the INT-CTXT game**: After the sequential encryption oracle finalizes, L_seq is fixed (no further entries can be added to the sequential oracle's log), but the INT-CTXT game continues. When `encrypt_chunk_at` is co-exposed, L = L_seq ∪ L_par and L_par may continue to grow post-sequential-finalization — entries added by subsequent `encrypt_chunk_at` calls must still be included in the forgery exclusion set. A formal model that treats "L is fixed" as an invariant after sequential finalization would incorrectly admit forgeries against `encrypt_chunk_at` outputs produced post-finalization. In the sequential-only game (no `encrypt_chunk_at` oracle), L = L_seq is indeed fixed after finalization and this distinction does not arise. The INT-CTXT game in both cases continues — the adversary may still query both decryption oracles, now attempting to submit a forgery (chunk_index, tag_byte, ciphertext) ∉ L. The random-access decryption oracle remains fully active post-encryption-finalization (it

reads only persistent state), and the sequential decryption oracle remains active until it has itself finalized. A formal model that terminates the INT-CTXT game at encryption oracle finalization would incorrectly exclude these post-finalization forgery attempts, admitting a false proof of Property 2. This is the symmetric counterpart to the decryption-finalization note below. **Sequential decryption oracle termination**: The sequential decryption oracle is likewise a partial function after finalization — once a final chunk (tag_byte=0x01) has been successfully decrypted, the decryptor rejects all further `decrypt_chunk` calls (returning InvalidData, §15.3). A formal model must include a parallel decryptor-finalization transition rule: the linear state fact representing the sequential decryptor must be consumed (or moved to a terminal absorbing state) upon successful decryption of a final chunk. Without this rule, the model admits adversary sequential decryption queries beyond the finalized position — which is undefined in the construction and produces spurious counterexample traces for Properties 3 and 4 (Property 4's `is_finalized()` state becomes reachable-but-unconstrained in symbolic models that do not consume the linear fact). **Sequential finalization does not terminate the INT-CTXT game**: After the sequential decryption oracle enters its terminal state, the random-access decryption oracle (`decrypt_chunk_at`) remains active — finalization affects only the sequential oracle's linear state fact, not the persistent state shared with the random-access oracle. Property 2's INT-CTXT game continues until `Corrupt(StreamKey)` or game end. A formal model that treats sequential finalization as game-over would incorrectly exclude post-finalization adversary interactions via the random-access oracle, potentially admitting Property 2 violations that the sequential oracle would not catch. **Forced early finalization strategy**: The adversary may legally exhaust the sequential decryptor's live state by presenting a genuine final-chunk ciphertext (obtained from the encryption oracle) at the appropriate sequential position, intentionally triggering finalization. After that, the sequential decryptor is in its terminal absorbing state and Properties 3–4 are vacuously satisfied; Property 2 remains active via the random-access oracle. This strategy is in-scope, benign, and does not constitute an attack — a formal model's decryptor-termination rule must permit it without adding a spurious axiom that restricts the adversary from presenting honest final-chunk outputs to the sequential decryptor. For a stream with fresh key (no `Corrupt(StreamKey, P, stream_id)`) and fresh base nonce (no `Corrupt(RNG, P, t)` for t = the §15.1 base_nonce generation step at stream encryptor initialization): The `Corrupt(RNG)` condition at base_nonce generation is load-bearing specifically for Property 5 (multi-stream isolation) and the $(\Sigma M_i)^2/2^{193}$ birthday bound — it prevents adversarially-forced base_nonce collisions across streams. For single-stream Properties 1–4, this condition is vacuous: base_nonce is public (transmitted in the stream header, §15.1), so adversary knowledge or choice of base_nonce does not weaken IND-CPA or INT-CTXT. A formal model targeting only Properties 1–4 may omit the streaming `Corrupt(RNG)` oracle. **Composed-protocol freshness**: The two conditions above are sufficient for standalone Theorem 13 analysis (stream key and base nonce, no ratchet dependency). When Theorem 13 is composed with the ratchet layer — e.g., stream key k delivered as ratchet application-layer content — two additional freshness conditions apply that are not visible from this theorem in isolation: (1) `No Corrupt(MessageKey, P, i)` at the ratchet send position i where k was delivered, and (2) shared-key fact modeling when multiple parties hold the same k. See §15.1 (Cross-party key sharing and Indirect corruption path notes) and §8.6 (Composition paragraph) for the full composed-protocol preconditions.

Precondition (compression independence): When compression is enabled, it must be applied independently per chunk with no cross-chunk shared state, dictionary, or context. This is the structural precondition for the multi-chunk-to-per-chunk hybrid argument: without it, compressed plaintext in chunk i could influence the ciphertext length of chunk i+1, creating statistical dependencies that break the per-chunk AEAD reduction. The reference implementation satisfies this precondition (zstd is invoked per-chunk with no shared dictionary). A reimplementer who introduces a shared compression dictionary violates this precondition and must re-examine the confidentiality reduction.

1. **Confidentiality**: Each chunk's ciphertext is indistinguishable from random (IND-CPA). Reduces to XChaCha20-Poly1305 confidentiality under unique nonces (guaranteed by the injective `base_nonce XOR (index || tag_byte || padding)` construction). **Stream-length dependence**: The IND-CPA guarantee is parameterized by the stream length Q (number of chunks); when encrypt_chunk_at is co-exposed as a co-oracle, Q = Q_seq + Q_par — the total number of AEAD invocations across both encryption oracle types (the 2^{64} sequential maximum per §15.2 bounds Q_seq; Q_par is bounded only by the caller's distinct-(index, tag_byte)-pair budget). The aggregate advantage bound is Q × Adv_IND-CPA(XChaCha20-Poly1305) via the Q-step hybrid reduction (see Security loss note below); at 128-bit per-chunk security, this gives approximately 128 − $\log_2(Q)$ effective IND-CPA bits. INT-CTXT has no Q-factor loss (direct forgery reduction, no hybrid). A researcher or tool citing Theorem 13 as a security claim must specify Q — a 10-chunk stream provides ≈128-bit IND-CPA, a 2^{32}-chunk stream provides ≈96-bit IND-CPA, and a 2^{64}-chunk stream (the sequential maximum per §15.2) provides ≈64-bit IND-CPA while INT-CTXT remains at 128 bits. This IND-CPA/INT-CTXT asymmetry at high Q is a non-obvious property of the construction that does not arise in standard monolithic AEAD usage. The reduction operates in the nAE (nonce-based AE) model [Rog04]: base_nonce is transmitted in the cleartext stream header (§15.1), so the adversary observes all per-chunk nonces before any ciphertext is produced. XChaCha20-Poly1305's IND-CPA security does not require nonce secrecy — only nonce uniqueness. A CryptoVerif model must emit base_nonce to the adversary (Out(base_nonce)) in the StreamEncInit rule, before any IND-CPA challenge query; a model that treats base_nonce as secret at challenge time proves a strictly stronger and incorrect property. **Compression scoping**: When compression is enabled, the IND-CPA claim applies to the post-compression AEAD layer — confidentiality of the compressed chunk, not the original plaintext. Ciphertext lengths may leak information about plaintext compressibility (shorter ciphertext for highly compressible data). This length leakage is outside the IND-CPA game (which requires equal-length challenge messages) and is scoped out of Theorem 13. **Length-equalizing by construction**: The game is length-equalizing — the adversary submits a post-compression byte string of length ℓ, and both the b=0 branch (real AEAD ciphertext) and b=1 branch (encryption of a uniformly random ℓ-byte string) produce ciphertexts of length ℓ + AEAD_TAG_SIZE. Ciphertext length therefore does not distinguish the two branches; there is no trivial length distinguisher between b=0 and b=1 regardless of whether compression is enabled. This is why the equal-length requirement of the IND-CPA game is satisfied: the relevant length is the post-compression submitted length, not the original plaintext length. **CryptoVerif compression modeling**: The game description above handles this by having the adversary submit post-compression byte strings of equal submitted length directly (compression is a preprocessing step outside the game). A CryptoVerif model that includes the compression step within the model rather than pre-applying it must require the adversary's challenge pair (m_0, m_1) to satisfy $|\text{compress}(m_0)| = |\text{compress}(m_1)|$. A model that permits length-differing challenge pairs after compression introduces a trivial ciphertext-length distinguisher into the IND-CPA game and will report a false distinguisher — the ciphertext length reveals b regardless of the AEAD's security. Equivalently: the game's equal-length constraint applies to the post-compression plaintext (the AEAD input), not to the original plaintext. When compression is enabled, Theorem 13 makes no confidentiality claim about the original (pre-compression) plaintext — only about the post-compression byte string passed to the AEAD. A caller requiring a formal confidentiality guarantee for the original plaintext must treat the compression function as a secrecy-preserving transformation, which is not proved here and is not generally true for general-purpose compressors. The practical impact is limited: file transfer does not have attacker-controlled plaintext injection, so CRIME/BREACH-style adaptive compression attacks do not apply. A formal model should either disable compression or treat the compression function as a stateless per-chunk deterministic preprocessing step outside the AEAD security game (each chunk is compressed independently with no cross-chunk dictionary or context — the multi-chunk-to-per-chunk hybrid argument depends on this independence). **Decompression failure collapsing**: On the decryption path, decompression can fail post-authentication (corrupted compressed data, decompression bomb exceeding CHUNK_SIZE). The implementation collapses all post-AEAD decompression failures to the same error as AEAD failure — a single observable failure event. A formal model must mirror this collapse: introducing a distinct post-AEAD decompression failure event would create a 1-bit oracle that leaks whether the ciphertext was valid-AEAD-but-bad-compression, violating the single-failure-event discipline.

2. **Integrity** *(sequential and random-access interfaces)*: No PPT adversary can forge a chunk that passes AEAD authentication (INT-CTXT) on either decryption oracle. Reduces to XChaCha20-Poly1305 ciphertext integrity in the [Rog04] nAE CTXT notion — the INT-CTXT parallel to Property 1's IND-CPA reduction in the same [Rog04] framework: a (nonce, aad, ciphertext) triple not in the encryption oracle log that a decryption oracle accepts, with security requiring only nonce uniqueness (not nonce secrecy). The nonce-respecting adversary restriction from [Rog04] §4 applies to CTXT for the same reason it applies to IND-CPA: §15.2 nonce injectivity is the shared precondition under which both notions hold. **Per-chunk INT-CTXT**: The INT-CTXT notion used here is per-chunk: each (nonce, aad, ciphertext) triple is authenticated independently — a forgery is a single triple (chunk_index, tag_byte, ciphertext) $\notin$ L that a decryption oracle accepts, where nonce and aad are derived per-chunk from (chunk_index, tag_byte, base_nonce). This is distinct from monolithic INT-CTXT, which authenticates the entire message as one unit. Per-chunk INT-CTXT is per-chunk-stronger (each individual chunk is forgery-resistant without requiring the full stream) and per-stream-weaker (the adversary could truncate the stream — omit the final chunk — without violating per-chunk INT-CTXT; Property 4 separately addresses this at the stream level). The per-chunk plaintext delivery semantics (§15.3) follow directly from per-chunk INT-CTXT: each chunk's plaintext is available after that chunk's authentication without waiting for the stream to complete.

3. **Ordering** *(sequential interface only, §15.3)*: A chunk encrypted at index i, when decrypted by a sequential decryptor at position next_index ≠ i, produces a mismatched nonce and AAD — the decryptor derives the chunk index from its own next_index (not from the chunk) and reads only tag_byte from the wire format — causing AEAD verification to fail. Ordering thus reduces to INT-CTXT under the injective nonce construction (§15.2), not to an explicit index comparison guard. §15.2 nonce injectivity is the load-bearing lemma for Property 3: distinct sequential positions i ≠ j produce distinct nonces (by the XOR-bijection argument), so the AEAD tag verification fails at position j when a chunk encrypted at i is presented. §15.4 AAD injectivity (chunk_index in the AAD) provides independent defense-in-depth but is not required for the Property 3 reduction — nonce mismatch alone causes AEAD failure. A proof of Property 3

needs only §15.2; Property 4's reduction is distinct and does require both §15.2 (tag_byte in nonce mask position 8) and §15.4 (tag_byte in AAD). **Symbolic vs. computational proof structure**: In a computational model (CryptoVerif), Property 3 requires INT-CTXT: accepting a ciphertext produced at (nonce_i, AAD_i) under (nonce_j, AAD_j) ≠ (nonce_i, AAD_i) is a forgery in the nAE INT-CTXT game (the triple (nonce_j, AAD_j, ciphertext_i) was not produced by the encryption oracle), and the adversary's advantage is bounded by Adv_INT-CTXT. In a symbolic model (Tamarin/ProVerif), Property 3 holds axiomatically from the AEAD encoding: symbolic decryption with wrong nonce or AAD is ⊥ by the AEAD constructor equations — no PPT adversary quantification or advantage bound is needed. A Tamarin modeler can prove Property 3 as a reachability lemma ("no trace reaches sequential decryptor accepting at position j for a chunk produced at i ≠ j") directly from the nonce-injectivity encoding, without constructing an INT-CTXT reduction. Property 4's cryptographic direction (no false finalization via forgery) is more complex in both models: even in Tamarin, the modeler must express that the adversary's free-constructor ciphertext cannot pass the AEAD verification rule, which is the symbolic non-forgery axiom. Scope accordingly: Property 3 ⇒ structural reachability lemma in Tamarin; Property 4 negative direction ⇒ non-forgery lemma in both models. The random-access interface (`decrypt_chunk_at`) accepts an explicit caller-provided index and does not enforce ordering — reordering is by design. **Winning condition**: The adversary wins Property 3 if the sequential decryption oracle accepts (tag_byte, ciphertext) at oracle position next_index = j, where either encryption oracle (sequential or random-access) produced that (tag_byte, ciphertext) at index i ≠ j (the security argument is the same — nonce injectivity at §15.2 causes AEAD failure at position j regardless of which oracle produced the chunk at index i). This is an INT-CTXT sub-game win: the (chunk_index=j, tag_byte, ciphertext) triple is in L (produced by an encryption oracle at index i), but the AEAD verification at position j uses nonce and AAD derived from j — which differ from those at i by the injectivity of §15.2 — so the decryptor's success constitutes a forgery under the INT-CTXT game at position j. The tag_byte is held fixed in this winning condition; attacks that also flip tag_byte (e.g., presenting a final-chunk ciphertext as non-final at the same position) are addressed by Properties 2 and 4 respectively, with reductions that additionally invoke §15.2 mask position 8 and §15.4 tag_byte injectivity. **Honest advancement is allowed**: Advancing the sequential decryptor via honest oracle queries (presenting genuine encryption oracle outputs at positions 0..j-1 to reach next_index = j) is legal adversary behavior and does not constitute a win. The win condition requires AEAD acceptance at position j of a chunk that was encrypted at position i ≠ j — not of a chunk encrypted at position j (which would be legitimate decryption). A Tamarin model should have no restriction on the adversary presenting honest ciphertexts in sequence to advance the decryptor's counter; the ordering property must hold regardless of how the decryptor arrived at position j. **Replay subsumption**: Property 3 implicitly subsumes replay resistance — a replayed chunk at index j where next_index > j is a special case of reordering and is rejected by the same nonce/AAD mismatch mechanism. The sequential counter's strict advance prevents acceptance of any previously-decrypted chunk index. This establishes the combined ordering + no-replay guarantee of [BKN04]'s stAE notion for the sequential interface; a formal reviewer mapping Theorem 13's properties to [BKN04]'s property list should treat replay resistance as covered here.

4. **Truncation resistance** *(sequential interface only, §15.3)*: Omitting the final chunk (tag_byte=0x01) is detectable — no non-final chunk satisfies the finality check. The sequential decryptor's `is_finalized()` accessor exposes stream completeness to the caller. The random-access interface does not track finalization state — truncation detection is the caller's responsibility when using random-access decryption. Property 4 has two distinct layers: (a) **State machine guarantee (positive direction)**: `is_finalized()=true` iff a final-chunk ciphertext (tag_byte=0x01) was successfully decrypted — a direct consequence of the `finalized` state transition in §15.3. This is axiomatic in a formal model: the decryptor's linear state rule sets `finalized=true` only on successful decryption of a final-tagged chunk, and the finalization accessor reads that flag. No proof obligation exists for this direction beyond correctly modeling the state machine. (b) **Cryptographic guarantee (negative direction)**: No adversary can produce false finalization without AEAD forgery. This is the direction that requires a proof: reaching `is_finalized()=true` via a non-genuine final chunk would require accepting a forged (chunk_index, tag_byte=0x01, ciphertext) not in L. Reduces to: INT-CTXT + nonce/AAD injectivity over tag_byte (§15.2, §15.4) — no non-final chunk produces a final-chunk nonce or AAD. **Winning condition**: The adversary wins Property 4 if the sequential decryptor reaches `is_finalized()=true` without the encryption oracle having submitted a query with is_last=true at the decryptor's expected terminal position — i.e., apparent stream completion without the genuine final chunk. This requires either forging a final-chunk ciphertext (excluded by INT-CTXT) or causing a non-final chunk (tag_byte=0x00) to be accepted as final (excluded by INT-CTXT

    - tag_byte injectivity over nonce and AAD at §15.2 mask position 8 and §15.4). **Prefix-safety subsumption**: Property 4 implicitly subsumes prefix-safety — an adversary holding honest chunks 0..k-1 who forges a final chunk at position k-1 (tag_byte=0x01) is excluded by INT-CTXT (the forged triple is not in L but is accepted by the decryptor, satisfying the Property 2 winning condition). Stream completion requires the genuine final-chunk ciphertext at its correct position. **Stream integrity corollary (Properties 3 + 4)**: `is_finalized()=true` from the sequential decryption interface iff the decryptor received exactly the chunks at indices 0..N in the correct order, where N is the index of the genuine final chunk. **Both directions**: (→) as established above — is_finalized()=true without receiving exactly 0..N in order is a win for Property 3 or 4; (←) if all genuine chunks 0..N were received in order (including the genuine final chunk at N), Property 4(a) guarantees that the decryptor transitions to finalized=true upon successfully decrypting the final-tagged chunk — no adversarial action is needed and no proof obligation is required beyond the state machine axiom. The ← direction is axiomatic from the §15.3 state machine (finalized=true is set iff a final-chunk AEAD passes), not a reduction. Tamarin modelers encoding this corollary as a biconditional lemma should note that the → direction requires the full INT-CTXT reductions for Properties 3 and 4, while the ← direction is an axiom (the state machine rule's conclusion). Encoding only → produces a sound but incomplete lemma; encoding iff correctly characterizes the finalization flag's precise semantic correspondence to stream completeness. Neither property alone establishes this: Property 3 prevents reordering but does not constrain which index finalizes the stream; Property 4 prevents false finalization but does not constrain the ordering of preceding chunks. Together, Property 3 ensures all preceding chunks 0..N-1 arrived at their correct sequential positions (any reordering would have failed as a nonce/AAD mismatch), and Property 4 ensures the final chunk at index N is genuine. A Tamarin stream-completion lemma should state the conjunction: "if StreamDecState(finalized=true, next_index=N+1), then for all i in 0..N, the chunk at index i was produced by the encryption oracle at index i." This compound guarantee corresponds to what [BKN04] calls strong integrity in stAE (ordering + no-replay + completeness) — though Property 4's cryptographic finalization exceeds stAE's truncation model (see §15 intro for the stAE scope boundary). **Combined security bound**: Properties 3 and 4 are proved via separate INT-CTXT reductions with distinct winning conditions and distinct reduction mechanisms (nonce mismatch for Property 3; nonce/AAD injectivity over tag_byte for Property 4). The conjunctive guarantee therefore has combined adversarial advantage at most Adv_Property3 + Adv_Property4 ≤ 2 × Adv_INT-CTXT(XChaCha20-Poly1305) by a union bound. A Tamarin lemma proving the conjunction `(is_finalized()=true ∧ ordering holds for all preceding chunks)` inherits this doubled bound; a proof that decomposes the conjunction into two separately-proved lemmas may cite each property's individual Adv_INT-CTXT bound. The 2× factor arises only from the conjunction and is absent from either property individually — it is not a weakness of the construction, but the standard union-bound overhead of proving two independent security properties via the same underlying primitive.

5. **Cross-stream isolation** *(under distinct stream_id = (key, base_nonce), §8.2)*: A chunk from stream A cannot be spliced into stream B. For same-key streams, isolation is provided by the AAD mechanism: different `base_nonce` values cause AEAD failure (base_nonce distinctness holds with overwhelming probability — see multi-stream nonce uniqueness note below for the $(\Sigma M_i)^2/2^{193}$ birthday bound). For different-key streams, isolation holds trivially by AEAD key mismatch regardless of base_nonce — the AAD mechanism is not required. A formal model parameterizing Property 5 on base_nonce alone correctly captures the same-key case; the full precondition is distinct stream_id (key or base_nonce differs). Reduces to: INT-CTXT + base_nonce binding in AAD (§15.4). **Winning condition**: The adversary wins Property 5 if either decryption oracle, initialized with stream_id_B = (key_B, base_nonce_B), accepts a (chunk_index, tag_byte, ciphertext) triple produced by an encryption oracle under stream_id_A ≠ stream_id_B. **Forgery representation bridge**: In the embedded single-stream INT-CTXT game used for the same-key reduction, the forgery set L_AEAD consists of (nonce, AAD, ciphertext) triples keyed at the AEAD level, not just ciphertext values. A chunk produced at (nonce_A, AAD_A, ciphertext) under stream A is absent from L_AEAD at (nonce_B, AAD_B) — the cross-stream presentation at stream B's parameters is an unconditional forgery against the embedded INT-CTXT challenger even though the ciphertext was honestly produced. A reader familiar with the per-stream forgery definition (which keys on chunk_index and tag_byte via L_seq or L_par) should not conclude that membership in L_A prevents a Property 5 win: L_A membership means the ciphertext was honestly produced; the Property 5 forgery is the fact that stream B's decryptor accepted it despite it never having been produced under stream B's AEAD parameters. **INT-CTXT reduction (same-key case)**: Embed both streams A and B into a single INT-CTXT game under key k = key_A = key_B. The reduction proceeds in two cases by how the cross-stream attack is presented. **Same-position case** (adversary presents a stream-A ciphertext encrypted at (i, t) to stream B's decryptor at the same position (i, t)): The nonces differ — base_nonce_A ≠ base_nonce_B, so nonce_A(i,t) = base_nonce_A XOR mask(i,t) ≠ base_nonce_B XOR mask(i,t) = nonce_B(i,t) by the §15.2 XOR-injectivity argument. AEAD acceptance requires the correct nonce, so acceptance at the mismatched nonce_B is a forgery. **Different-position case** (adversary presents a stream-A ciphertext encrypted at (i, t_A) to stream B's decryptor at a different position (j, t_B)): The nonces at different positions may coincide by birthday collision — this case cannot be handled by the nonce argument alone. It is handled unconditionally by the AAD argument: AAD_A contains base_nonce_A as a field (§15.4), and AAD_B contains base_nonce_B. Since base_nonce_A ≠ base_nonce_B, AAD_A ≠ AAD_B regardless of position. The (nonce_B, AAD_B, ciphertext) tuple was never output by the INT-CTXT encryption oracle (which only produced tuples with AAD_A), so AEAD acceptance at (nonce_B, AAD_B) is

an unconditional forgery. The reduction in both cases is direct: the adversary's cross-stream acceptance is forwarded as the forgery. **Reduction structure note**: The AAD argument provides the unconditional component of the reduction (no birthday term needed); the nonce argument provides the same-position proof more directly. A formal proof of Property 5 should use the AAD argument as the primary reduction mechanism and note the nonce argument handles the same-position case efficiently. The §15.4 base_nonce table entry reflects that both bindings are present; only the AAD binding is unconditionally load-bearing for the formal reduction. **INT-CTXT reduction (different-key case)**: The stream-B INT-CTXT game uses key_B. A ciphertext produced under key_A, when submitted to AEAD.Decrypt(key_B, ...), is a (nonce, ciphertext) pair not in key_B's encryption oracle log — it was produced under a different key. This is directly a forgery in the INT-CTXT game at key_B. No AAD argument is needed.

Non-goal (forward secrecy): Theorem 13 makes no forward secrecy claim. The stream key is a static caller-provided value with no rotation mechanism — there is no KEM ratchet step or key evolution analogous to the LO-Ratchet. Zeroization on drop (§15.1) is an implementation-level hardening measure and does not imply cryptographic FS. `Corrupt(StreamKey)` at any point retroactively breaks all chunk confidentiality and integrity: the adversary can decrypt all previous ciphertexts and forge new ones. Applications requiring FS must rotate the stream key at the application layer and use a fresh stream instance per rotation epoch. This is a design constraint, not a gap: the streaming layer is intended to be composed with the ratchet layer (which provides FS via KEM ratchet steps), and FS is delegated to the ratchet. A standalone streaming deployment without a FS-providing outer layer has no FS guarantee.

No novel cryptographic assumptions are introduced. The construction composes standard AEAD with deterministic nonce derivation. **Multi-instance AEAD relationship**: Property 5 is a *cross-stream binding* property — it prevents a chunk from stream A from being accepted by stream B's decryptor. It is distinct from *multi-instance (MI) confidentiality* [BT16], which bounds the joint adversarial advantage across N concurrent streams. MI-IND-CPA for this construction follows from applying Property 1 independently per stream (advantage additive across streams by a union bound), plus the additive $(\Sigma M_i)^2/2^{193}$ birthday term for cross-stream nonce collision under a shared key (see multi-stream nonce uniqueness note below). **Explicit MI-IND-CPA formula (N streams sharing key k)**: $\text{Adv}_{\text{MI-IND-CPA}} \leq Q_{\text{total}} \times \text{Adv}_{\text{IND-CPA}}(\text{XChaCha20-Poly1305}) + Q_{\text{total}}^2/2^{193}$, where $Q_{\text{total}} = \Sigma_i M_i$ is the grand total AEAD invocations across all N streams. The per-stream hybrid losses sum to $Q_{\text{total}} \times \text{Adv}_{\text{IND-CPA}}$ (not $\Sigma_i (M_i \times \text{Adv}_{\text{IND-CPA}})$ per stream separately — those are equal, but the birthday term is over Q_{total}^2, not $\Sigma_i M_i^2$). The common error is computing the birthday term per-stream and summing, obtaining $\Sigma_i M_i^2/2^{193}$ instead of $(\Sigma_i M_i)^2/2^{193}$ — these differ by $2 \times \Sigma_{i \neq j} M_i M_j$ (the cross-stream collision terms). The $Q \leq 2^{32}$ recommendation applies to Q_{total} across all same-key streams, not per-stream: a deployment with N streams each using $2^{32}/N$ chunks has $Q_{\text{total}} = 2^{32}$ and ≥ 96-bit IND-CPA regardless of N; a deployment with $N = 2^{32}$ streams each using 1 chunk also has $Q_{\text{total}} = 2^{32}$ and identical security. Property 5 and MI-confidentiality together give the full multi-stream security picture: Property 5 ensures cross-stream authenticity (no splicing), and MI-IND-CPA ensures cross-stream confidentiality (no cross-stream plaintext leakage). A researcher situating Theorem 13 against the MI-AEAD literature [BT16] should note that Theorem 13's per-stream security statements imply MI security with the additive birthday overhead. **Security loss (multi-chunk)**: The two properties use structurally distinct reductions. For Property 1 (IND-CPA), the multi-chunk adversary reduces to per-chunk IND-CPA via a Q-step hybrid: one chunk is switched from real to random at each step, each step incurs at most $\text{Adv}_{\text{IND-CPA}}$ loss, giving aggregate advantage at most $Q \times \text{Adv}_{\text{IND-CPA}}(\text{XChaCha20-Poly1305})$. **Hybrid order-independence (co-oracle case)**: When `encrypt_chunk_at` is co-exposed ($Q = Q_{\text{seq}} + Q_{\text{par}}$), the adversary may submit queries in arbitrary order — index 500 before the sequential oracle reaches index 3, for example. The Q-step hybrid works regardless of query order because the per-chunk IND-CPA challenges are independent: each (index, tag_byte) pair uses a unique (key, nonce, AAD) triple by §15.2 injectivity, so switching one chunk from real-to-random does not affect the distribution of any other chunk's oracle response. The prover's switching schedule can be any fixed enumeration of the Q distinct (index, tag_byte) pairs — it need not follow the adversary's query order. In EasyCrypt or CryptoVerif, this justifies treating the Q queries as Q independent IND-CPA experiments regardless of how they interleave across the two encryption interfaces. **Concrete security bound**: For XChaCha20-Poly1305 at 128-bit per-chunk security ($\text{Adv}_{\text{IND-CPA}} \approx 2^{-128}$), a Q-chunk stream provides approximately $128 - \log_2(Q)$ bits of IND-CPA security. At $Q = 2^{64}$ (the sequential stream's maximum chunk count per §15.2), IND-CPA security degrades to ≈ 64 bits while INT-CTXT remains at ≈ 128 bits — a significant asymmetry. Recommended safe operating limit: $Q \leq 2^{32}$ chunks per sequential stream gives ≥ 96-bit IND-CPA. A CryptoVerif model should include Q as a session parameter and state the concrete IND-CPA bound as a function of Q. For Property 2 (INT-CTXT), no hybrid is needed: the adversary produces a single forgery (chunk_index*, tag_byte*, ciphertext*) ∉ L that some decryption oracle accepts. This is directly a forgery in the per-chunk nAE INT-CTXT game at nonce base_nonce XOR mask(chunk_index*, tag_byte*) and aad* — the reduction runs all Q encryption oracle queries against the nAE INT-CTXT challenger (which accepts arbitrary distinct nonces) and forwards the adversary's forgery directly. No Q-factor loss occurs. The aggregate INT-CTXT advantage is $\text{Adv}_{\text{INT-CTXT}}(\text{XChaCha20-Poly1305})$, not $Q \times \text{Adv}_{\text{INT-CTXT}}$. A formal reviewer should not expect a Q-fold INT-CTXT security loss — the hybrid argument is IND-CPA-specific. The [BN00] IND-CPA + INT-CTXT ⇒ IND-CCA composition applies per-chunk after both reductions.

Multi-stream nonce uniqueness: The nonce injectivity argument (§15.2) holds within a single stream (fixed base_nonce). Across N streams sharing the same key with M_i chunks each, nonce uniqueness depends on the absence of cross-stream nonce collisions. The IND-CPA reduction requires all nonces across all AEAD invocations under the same key to be distinct; the birthday bound on 192-bit nonces gives collision probability $\sim(\Sigma M_i)^2/2^{193}$ over the total number of AEAD invocations. **Birthday bound derivation**: M_i is the total AEAD invocation count for stream i across both interfaces — $M_i = Q_{\text{seq\_i}} + Q_{\text{par\_i}}$, counting every non-final and final chunk (every call uses the nonce space regardless of tag_byte). $M = \Sigma M_i$ is the grand total under the same key. The birthday bound for a collision among M nonces drawn from a 2^{192}-element space is $P(\text{any collision}) \leq M(M-1)/2 \,/\, 2^{192} < M^2/2^{193}$ (applying $N^2/2^{\{bits+1\}}$ with bits=192: $N^2/(2 \times 2^{192}) = N^2/2^{193}$). The bound applies only when streams share the same 32-byte key — streams under different keys have independent AEAD, and a nonce coincidence across keys is not a collision in any security game. For any practical workload, this is negligible (e.g., 2^{64} total chunks yield collision probability $\sim 2^{-65}$). Due to the mask structure — index (8 bytes) ‖ tag_byte (1 byte) ‖ 0x00×15 — cross-stream collisions are batch-correlated: any collision requires $\Delta[9{:}24] = 0$ (the 120-bit zero-padding region of both base_nonces must align, probability 2^{-120} per stream pair). Once this holds, collisions occur at all chunk positions where $\text{index\_A} \oplus \text{index\_B} = \Delta[0{:}8]$ and $\text{tag\_A} \oplus \text{tag\_B} = \Delta[8]$, producing up to $\min(M_A, M_B)$ simultaneous collisions — not a single-chunk event. The zero-padding in the mask is a deliberate design choice that enforces this batch-correlation structure: by fixing the low 15 mask bytes to zero, any cross-stream nonce collision requires $\Delta[9{:}24] = 0$ (probability 2^{-120} per stream pair). A mask using all 24 bytes for varying data would permit targeted single-chunk collisions at cost 2^{-64} per attempt (requiring only $\Delta[0{:}8]$ alignment); the zero-padding raises the targeted attack floor by 2^{56}, ensuring any adversary seeking a cross-stream collision must first solve a 2^{-120} per-pair problem before any chunk-level collision is possible. A base_nonce collision ($\Delta = 0$) is the maximally catastrophic case: all corresponding (index, tag_byte) positions collide, with total security loss for both streams. The IND-CPA reduction for a multi-stream adversary incurs an additive $(\Sigma M_i)^2/2^{193}$ security loss term. A formal model may choose either: (a) a single-key multi-stream game with the additive birthday term, or (b) a per-stream freshness assumption (each stream uses a distinct key) under which cross-stream nonce collisions are harmless. **Birthday denominator pitfall**: The 2^{193} denominator is the full XChaCha20 nonce space — 2×2^{192} because the birthday bound on N items is $N^2/2^{\{bits+1\}}$. A modeler who notices that only 9 bytes of the mask vary within a stream might use 2^{73} as the denominator (the variable-bits space), obtaining $(\Sigma M_i)^2/2^{73}$ — a factor-2^{120} error. The 72-bit variable space is the within-stream per-sequential-stream index budget (the per-stream nonce budget from §15.2), not the uniqueness domain for the birthday argument. Nonce uniqueness in the birthday collision calculation operates over the full 192-bit nonce (XOR of base_nonce and mask), so the correct denominator for cross-stream collision probability is always 2^{193}. **Q_{par} contribution to ΣM_i**: The `encrypt_chunk_at` interface accepts the full u64 index domain (0..u64::MAX) — the sequential encryptor's $0..2^{64}$-2 bound (§15.2) does not apply to random-access encryption. Combined with tag_byte ∈ {0x00, 0x01}, a single stream using only the random-access oracle can contribute up to 2^{65} distinct (index, tag_byte) pairs to ΣM_i — exceeding the 2^{64} sequential maximum by a factor of 2. A modeler who bounds each stream's $M_i \leq 2^{64}$ based on the sequential cap would undercount ΣM_i when random-access queries are present. The $(\Sigma M_i)^2/2^{193}$ formula remains correct in all cases — the denominator 2^{193} is the full XChaCha20 nonce space — but the total invocation count in ΣM_i must include Q_{par} from all random-access calls, not be bounded by the sequential stream length.

Lemma (KDF_Root output independence): For any invocation of KDF_Root(rk, ss) → (rk', ek) — whether at KEX initialization (§4.3 Step 3) or at a KEM ratchet step (§5.2 Step 3) — the two 32-byte output halves rk' and ek are computationally independent and individually indistinguishable from uniform, provided: (i) the IKM has sufficient min-entropy (at least one uncompromised KEM shared secret), and (ii) the salt rk is either fixed (0^{32} at KEX) or pseudorandom (an HKDF output from a prior step, invoking the dual-PRF property). The independence argument: HKDF-Expand produces T(1) = HMAC(PRK, info ‖ 0x01) and T(2) = HMAC(PRK, T(1) ‖ info ‖ 0x02); knowing T(1) does not help predict T(2) because PRK is unknown and HMAC keyed by PRK is a PRF — a standard hybrid argument over the counter-mode blocks establishes this. This lemma is used inductively: the base case is KEX initialization (the Composition paragraph below), and the inductive step is each subsequent KEM ratchet step. Theorems 3, 4, 5, and 10 all depend on this property.

Corollary (Cross-epoch key independence): Epoch keys from distinct KEM ratchet steps are pairwise independent. At step i, KDF_Root(rk_i, ss_i) produces (rk_{i+1}, ek_i); at step $j \neq i$, KDF_Root(rk_j, ss_j) produces (rk_{j+1}, ek_j). Each ss_i is a fresh KEM shared secret from Encaps(pk_r) — under X-Wing IND-CCA2, ss_i is indistinguishable from uniform and independent of ss_j. Since the inputs differ across invocations (at minimum ss_i ≠ ss_j with overwhelming probability), the HKDF outputs (rk_{i+1}, ek_i) and (rk_{j+1}, ek_j) are pairwise independent under the PRF guarantee. This corollary, combined with the intra-invocation lemma above, underpins Theorem 3's claim that "each message is encrypted under a distinct message key": messages in different epochs use keys derived from independent epoch keys, and messages within the same epoch use distinct counters under the same epoch key.

Composition (LO-KEX → LO-Ratchet): If the LO-KEX session key is fresh (Theorem 1), then the initial ratchet state satisfies the preconditions of Theorems 3-5. Specifically: KDF_KEX (§2.3) produces (rk, ek) via a single HKDF call with 64 bytes of output split at the 32-byte boundary. Under the KDF_Root output independence lemma (above), rk and ek are computationally independent and indistinguishable from uniform, provided the IKM has sufficient min-entropy (guaranteed by at least one uncompromised KEM shared secret, per the LO-KEX freshness predicate). The first ratchet message uses ek directly as the epoch key (counter 0 is consumed by the session-init first message; the ratchet starts at counter 1) — no KEM ratchet step precedes this use, so PCS (Theorem 5) does not apply to the initial epoch. PCS guarantees begin at the first direction change, when a KEM ratchet step introduces a fresh shared secret independent of the KEX-derived material. Additionally, the init preconditions (§4.6) require $rk \neq 0^{32}$ and $ek \neq 0^{32}$; under the random oracle model for HKDF, a 64-byte output produces an all-zero 32-byte component with probability 2^{-256} per component, so freshness (indistinguishability from uniform) implies non-degeneracy with overwhelming probability. This establishes the base case of the inductive argument for invariants (j)-(m) (§5.1); subsequent KEM ratchet steps maintain non-degeneracy probabilistically (2^{-256} per step).

Corollary (LO-KEX → LO-Call): If the LO-KEX session key is fresh (Theorem 1), then call key secrecy (Theorem 8) holds for any call initiated within the resulting session. The argument is a two-hop reduction: (1) The LO-KEX → LO-Ratchet composition (above) establishes that the session root key rk is computationally indistinguishable from uniform under the KEX freshness conditions. (2) Theorem 8 requires only rk freshness as its precondition for call key secrecy; substituting the composition result from step 1 satisfies this precondition. The combined security loss is additive across both reduction steps (one X-Wing IND-CCA2 advantage term per step). A formal modeler proving "call keys are fresh given uncompromised identity keys" may apply this corollary directly, without separately composing Theorem 1 + Theorem 8. This corollary is the KEX→Call direction of the full transitivity in §9.10(a).

Composition (LO-Ratchet → Streaming AEAD): When the ratchet is used to deliver a stream key k for use with Theorem 13 (§15), the composition introduces two additional freshness conditions beyond those of the standalone streaming layer (§8.3 LO-Stream freshness paragraph) that are not visible from §8.6's ratchet theorems alone. A formal modeler composing the ratchet with the streaming layer must consult §15.1 for the full precondition set; in particular: (1) **Indirect corruption path** — if k is delivered as ratchet application-layer message content at send position i, the composed freshness predicate requires `No Corrupt(MessageKey, P, i)` (Corrupt(MessageKey) at position i reveals mk, which decrypts the delivery message and transitively reveals k, without any Corrupt(StreamKey) or Corrupt(RatchetState) query being issued; equivalently: §8.3 F1 must hold for the epoch containing send position i — see the §15.1 indirect corruption note for the formal translation between `No Corrupt(MessageKey, P, i)` and the F1 representation in §8.2's primitive oracle set). (2) **Shared-key fact modeling** — if both Alice and Bob hold the same stream key k (each initiating a distinct stream using k with different base_nonce values), a per-party freshness predicate is syntactically satisfied even when k is known to the adversary via a Corrupt query on the other party's stream. A composed model must represent the shared key with a shared-knowledge fact (e.g., `!StreamKeyShared(k, [Alice, Bob])`) rather than two independent per-party secrets. Both conditions are elaborated with formal model guidance in §15.1.

Forward secrecy timeline (KEX → Ratchet handoff): KEX-level and ratchet-level forward secrecy activate at different points in a session's lifecycle:

Phase	Forward secrecy source	Dependency	Subsumed when
Session establishment (§4.3-4.4)	KEX-level: secrecy depends on sk_SPK_B (and sk_OPK_B if used) not being corrupted	Theorem 1 + §7.1 Multi-key FS	First KEM ratchet step completes
Initial epoch (pre-first-direction-change)	Same as above — no KEM ratchet step has occurred, so ratchet-level FS is not yet active	Theorem 1 only (PCS/Theorem 5 does not apply)	First KEM ratchet step
After first KEM ratchet step (send side)	Ratchet-level: ek_s overwritten, old send epoch key unrecoverable	Theorem 4 / Lemma 4a	Immediate (one step)
After first KEM ratchet step (receive side)	Partial: prev_ek_r retains old epoch key	Theorem 4 / Lemma 4b (counterexample)	Second KEM ratchet step (prev_ek_r overwritten)
After second KEM ratchet step	Full ratchet-level FS on both sides; KEX-derived material fully subsumed	Theorem 4 (both lemmas succeed)	—

The critical window is the initial epoch: forward secrecy depends entirely on KEX-level key deletion (sk_SPK_B, sk_OPK_B). Once the first KEM ratchet step completes, session secrecy no longer depends on sk_SPK — the ratchet's per-epoch FS takes over. This is relevant to §9.8 (SPK Rotation Lifecycle): the SPK grace period creates a window where KEX-level forward secrecy is not yet achieved for sessions using that SPK, but this window closes at the first KEM ratchet step regardless of SPK deletion timing.

15. Streaming AEAD

Section numbering note: §15 appears before §9 (Open Research Problems) and §10 (References) because it was appended after the main section sequence was established; §9 and §10 are intentional tail-anchors. The numbering reflects insertion order, not reading order.

Chunked AEAD with counter-derived nonces over XChaCha20-Poly1305. Each stream uses a single caller-provided 32-byte key and a random 192-bit base nonce. The construction is inspired by STREAM (Hoang, Reyhanitabar, Rogaway, Vizár, 2015) but uses counter-based nonce derivation for random-access decryption rather than ciphertext chaining. The security game for this construction (Theorem 13) is a dual sequential+random-access oracle game that does not coincide with any single prior AE notion: the sequential-only sub-game is closest to the stateful AE notion of Bellare, Kohno, and Namprempre [BKN04]; the random-access oracle is the novel element that enables index-addressable decryption, shifting truncation resistance and ordering from protocol-enforced properties to caller obligations. Properties 1–3 (confidentiality, integrity, ordering) fall within [BKN04]'s stAE coverage — [BKN04] §3 decomposes stAE into IND-CPA (Property 1), INT-CTXT (Property 2), and stateful ordering + no-replay (Property 3; see Replay subsumption note under Property 3). Note on mechanism divergence: [BKN04]'s ordering proof uses state synchronization — the encryptor and decryptor share a sequence counter, so a mismatched chunk is detected by counter mismatch at the decryptor. This construction uses §15.2 nonce injectivity instead, since the encryptor and decryptor maintain independent state (§15.3). The coverage is analogous (Property 3 establishes the same ordering + no-replay guarantee), but a formal reviewer cannot directly port the [BKN04] ordering proof path — the reduction mechanism differs. The Property 3 reduction should cite §15.2 nonce injectivity, not state-synchronization logic. Properties 4–5 (truncation resistance and cross-stream isolation) extend beyond stAE's scope and have no direct counterpart in [BKN04]. Property 4 is strictly stronger than [BKN04]'s implicit truncation model: stAE relies on the transport layer's "end of stream" signal for truncation detection (e.g., TCP connection close informs the receiver that no further data will arrive), leaving truncation detection to the transport rather than the cryptographic layer. This construction instead provides cryptographic finalization detection via `is_finalized()`: the flag is set only upon successful AEAD authentication of a genuine tag_byte=0x01 chunk, so a stream that delivers all N chunks and then closes is indistinguishable from one that was silently truncated at chunk N-1 in stAE — but distinguishable in this construction (the final-chunk tag must be present and authenticated). A transport-layer truncation cannot substitute for a genuine final-chunk ciphertext without AEAD forgery.

15.1 State

An encryptor/decryptor maintains:

- **key**: 32-byte XChaCha20-Poly1305 key (caller-provided, not derived from protocol state).
- **base_nonce**: 24-byte random nonce, generated once per stream via the OS CSPRNG. Public (transmitted in the stream header).
- **next_index**: u64, the next sequential chunk index. Starts at 0, incremented after each sequential encrypt/decrypt.
- **finalized**: boolean, set when a chunk with tag_byte=0x01 is processed (encrypt) or successfully decrypted (sequential decrypt only).
- **caller_aad**: variable-length application-supplied context (file ID, channel ID, etc.), provided once at stream init and constant across all chunks.
- **version**: u8, stream format version (currently 0x01).
- **flags**: u8, bit 0 = compression (zstd), bits 1-7 reserved (must be zero).

Stream keys are independent of ratchet state Σ — the key is caller-provided and not derived from rk or any epoch key.

The 26-byte stream header (version, flags, base_nonce) is transmitted in cleartext with no independent MAC. Header integrity is enforced implicitly: each chunk's AAD includes all header fields (§15.4), so header tampering causes every subsequent chunk's AEAD verification to fail. This is analogous to the ratchet header treatment (§8.1: headers are public and observable). A formal model's StreamDecryptInit rule should accept adversary-controlled header bytes — authentication occurs at the first DecryptChunk rule, not at initialization. (The implementation additionally rejects structurally invalid headers at init as defense-in-depth, returning two distinct observable error types: `UnsupportedVersion` for an unknown version byte, and `AeadFailed` for reserved flag bits set. A formal model need not replicate this two-type split since the flags and version fields in AAD (§15.4) enforce the same property at the first chunk; a model that unifies both as a single StructuralRejection is correct. For Channel 2 modeling (§9.1), note that `UnsupportedVersion` at init is a version-probe oracle — an adversary sending a header with version=0x02 learns whether the decryptor supports it, without triggering an AEAD oracle. `AeadFailed` for reserved flags is less informative. Both are pre-AEAD structural failures that expose no key material.)

Cross-party key sharing (composed protocol): In the primary usage pattern — ratchet delivers a file key k to both Alice and Bob; both parties independently construct a stream using (key=k, base_nonce=...) — Alice and Bob share the same stream key k. The streaming AEAD freshness predicate is per-party: `No Corrupt(StreamKey, Alice, stream_id_A)` and `No Corrupt(StreamKey, Bob, stream_id_B)` are independent conditions. However, if Alice's stream key is corrupted, k is revealed, transitively compromising Bob's stream (same k, different base_nonce) even though no Corrupt query was issued for Bob's stream — the predicate is syntactically satisfied but the key is known to the adversary. A formal model of the composed system (ratchet → file key → streaming) must represent the shared key with a shared-knowledge fact (e.g., `!StreamKeyShared(k, [Alice, Bob])`) so that a Corrupt query on either party's stream reveals k to the adversary as a shared value, not as two independently-scoped secrets. The per-party freshness predicate is correct for the standalone streaming layer (no key sharing assumed); key sharing is a composed-protocol obligation. **Indirect corruption path via ratchet delivery**: In the primary usage pattern, k is delivered as ratchet application-layer message content, encrypted under the ratchet message key mk at send position i. In this case, the composed freshness predicate additionally requires no Corrupt(MessageKey, P, i) — corruption of the ratchet message key at position i reveals mk, which decrypts the delivery message and transitively reveals k, compromising all streams under k without any Corrupt(StreamKey) query being issued. This condition is not implied by the per-party stream freshness predicate (`No Corrupt(StreamKey, P, stream_id)`) or by the ratchet-layer freshness predicate (`No Corrupt(RatchetState, P, t)`) at the granularity that standalone streaming models typically expose. A composed Tamarin or ProVerif model must include `No Corrupt(MessageKey, P, i)` as an additional lemma precondition when the stream key is a ratchet-delivered value, where i is the delivery position of k in the ratchet send chain. **Translation to §8.2 primitives**: `Corrupt(MessageKey, P, i)` does not appear as a primitive oracle in §8.2's corruption table (which lists Corrupt(IK), Corrupt(SPK), Corrupt(OPK), Corrupt(RatchetState), Corrupt(CallKeys), Corrupt(StreamKey), Corrupt(RNG)). It is implied by `Corrupt(RatchetState, P, t)` for any time point t within the cryptographic epoch containing send position i — corrupting the ratchet state reveals the epoch send key ek_s, from which mk_i = KDF_MsgKey(ek_s, i) is directly derivable. Equivalently, the condition `¬Corrupt(MessageKey, P, i)` holds iff §8.3 F1 holds for the epoch containing delivery position i (`¬Corrupt(RatchetState, P, t)` for any t during that epoch). A formal model using §8.2's primitive oracle set should replace `No Corrupt(MessageKey, P, i)` with this F1 condition; a model that omits it admits `Corrupt(RatchetState)` as an implicit path to k without being flagged as a freshness violation.

Key commitment: XChaCha20-Poly1305 does not provide key commitment (§8.4). This is not a concern for streaming AEAD: each stream uses a single caller-provided key with no trial decryption — the same reasoning as §8.4's key-commitment note for the ratchet (epoch identification selects a single key before decryption).

15.2 Nonce Derivation

Per-chunk nonce is derived by XORing a 24-byte mask into the base nonce:

```
mask = chunk_index  (8 bytes, big-endian u64)
    || tag_byte     (1 byte: 0x00 = non-final, 0x01 = final)
    || 0x00 * 15    (15 zero bytes, padding)

chunk_nonce = base_nonce XOR mask
```

Injectivity: For two chunks (i_1, t_1) and (i_2, t_2), $\text{mask}_1 = \text{mask}_2$ iff $i_1 = i_2$ and $t_1 = t_2$. Since XOR with a constant (base_nonce) is a bijection, distinct (index, tag_byte) pairs always produce distinct nonces within a single stream. The counter-based XOR-nonce derivation (XOR-ing a counter-derived mask into a base nonce) is a standard technique in the nAE (nonce-based AE) framework [Rog04]: per-chunk IND-CPA security reduces to unique nonces under the same key, which the injectivity argument above guarantees. No additional security assumption is required beyond nonce uniqueness and key secrecy — nonce secrecy is not needed (base_nonce is public). **Effective nonce variation budget**: Only 9 of the 24 nonce bytes vary within a single stream — 8 bytes for chunk_index (positions 0–7) and 1 byte for tag_byte (position 8). The remaining 15 bytes (positions 9–23) are always zero in the mask (see layout above), so they take their value directly from base_nonce and are constant across all chunks in the stream. The maximum number of distinct nonces per sequential stream is therefore 2^{64} (indices $0..2^{64}-2$ with tag_byte=0x00, plus the one final-chunk nonce with tag_byte=0x01 at the last index). The 120 fixed bits of base_nonce do no within-stream nonce-uniqueness work — they are entirely load-bearing for cross-stream isolation (ensuring streams under the same key have non-overlapping nonce spaces, per the multi-stream birthday analysis) and enforce a batch-correlation property that raises the targeted cross-stream attack floor: any cross-stream nonce collision requires those 15 fixed-zero mask bytes to align across both base_nonces (probability 2^{-120} per stream pair), raising the attack floor from the 2^{-64} that a fully-varying mask would permit; see the multi-stream birthday note's "batch-correlation" paragraph for the full security rationale. A CryptoVerif model computing the Q-chunk hybrid security loss should bound oracle queries against the 2^{64} per-stream budget for the sequential-only case ($2^{64} - 1$ non-final nonces plus one final nonce = 2^{64} distinct nonces per sequential stream), not the full 2^{192} XChaCha20 nonce space. When `encrypt_chunk_at` is co-exposed as a co-oracle, the per-stream budget extends to up to 2^{65} distinct (index, tag_byte) pairs (full u64 × {0x00, 0x01}); a CryptoVerif model that bounds $Q \le 2^{64}$ for the co-oracle case understates the parallel oracle's nonce budget by 1 bit. See the multi-stream birthday note's "Q_par contribution to ΣM_i" paragraph for the parallel-oracle correction.

The tag_byte in mask position 8 makes the nonce space disjoint between final and non-final chunks at the same index, preventing an attacker from stripping the final-chunk marker and appending additional chunks.

The chunk index is bounded for the sequential interfaces: ChainExhausted fires when next_index reaches $2^{64} - 1$ (the guard fires before use, so the maximum usable sequential index is $2^{64} - 2$, giving $2^{64} - 1$ total chunks per sequential stream — paralleling the ratchet's counter exhaustion guard in §5.3). The parallel encryption interface (`encrypt_chunk_at`, §15.3) has no ChainExhausted guard and accepts the full u64 index domain (0..u64::MAX); callers are responsible for staying within their intended range. Nonce injectivity holds for all reachable indices across both interfaces. A symbolic model using unbounded successor terms must either bound the index or add an explicit overflow axiom — without one, the model admits unreachable index values whose big-endian encoding wraps modulo 2^{64}, silently producing nonce collisions with smaller indices and breaking the injectivity guarantee.

15.3 Encrypt / Decrypt Interfaces

The construction provides two encryption interfaces and two decryption interfaces with different state semantics:

Sequential (`encrypt_chunk` / `decrypt_chunk`): Stateful — advances next_index, sets finalized on tag_byte=0x01 (encrypt) or successful decryption of a final chunk (decrypt). The chunk index is implicit (derived from next_index). Ordering and truncation detection are enforced by construction: the decryptor's internal counter must match the chunk's position in the AAD, and is_finalized() reflects whether the final chunk has been seen.

Random-access encryption (`encrypt_chunk_at`): Stateless — takes an explicit index parameter, reads only persistent state (key, base_nonce, version, flags, caller_aad), does not advance next_index, and has no ChainExhausted guard (accepts the full u64 index domain, including u64::MAX). No post-finalization guard. The caller must ensure that (index, tag_byte) pairs are distinct across all calls on the same stream instance, including any sequential encryption calls on the same instance — violating this constraint reuses the same nonce under the same key, breaking IND-CPA with an immediate distinguisher.

Random-access decryption (`decrypt_chunk_at`): Stateless — takes an explicit index parameter, does not advance next_index, does not set finalized regardless of tag_byte. The caller controls the decryption order and must track completeness externally. AEAD authentication still validates each chunk independently (correct key, correct nonce for the given index, correct AAD), but no inter-chunk ordering or stream-level truncation detection is provided.

Both interfaces are atomic: on failure (AEAD rejection, decompression failure, ChainExhausted, post-finalization), the encryptor/decryptor state is unchanged — next_index is not advanced and finalized is not set. A formal model's sequential rule must re-produce the original linear state fact on the failure branch, not an advanced one. (This parallels the ratchet's snapshot/rollback discipline in §5.4, but is simpler — no multi-field snapshot is needed since only next_index and finalized can change.)

Pre-AEAD framing guard: Both decryption paths (`decrypt_chunk` and `decrypt_chunk_at`) include a structural length check before AEAD is attempted — the guard is implemented in the shared inner helper and fires regardless of which interface is called: for uncompressed non-final chunks, the ciphertext must have length exactly STREAM_CHUNK_SIZE + AEAD_TAG_SIZE (AEAD overhead = 16 bytes); ciphertexts of incorrect length are rejected with `InvalidData` before any AEAD operation runs. This is a structurally-observable rejection distinct from AEAD failure: it occurs pre-AEAD (no authentication oracle is invoked), is based only on the observable ciphertext length (reveals no key material), and returns `InvalidData` rather than `AeadFailed` . A formal model should treat this as a structural pre-check rule in both the DecryptChunk and DecryptChunkAt model rules (analogous to the ratchet's deserialization structural checks in §8.4 / §9.4) — a model that includes the framing guard only in the sequential rule omits it from the random-access path and diverges from the implementation. The check is safe to model as observable from either interface: the code comment confirms it "does not reveal whether AEAD would have succeeded." The Channel 2 length-probing consideration applies to both interfaces equally: an adversary querying `decrypt_chunk_at` with short or oversized uncompressed non-final ciphertexts learns the guard outcome (InvalidData vs. AeadFailed) from either path without accessing the AEAD oracle.

Post-decompression size enforcement: For compressed non-final chunks, both decryption paths additionally enforce that the decompressed output is exactly STREAM_CHUNK_SIZE bytes after authentication succeeds; size mismatches are collapsed to `AeadFailed` . This is a second post-AEAD collapse point beyond general decompression failure: the error categories "decompressed to wrong size" and "zstd frame structurally invalid" are both returned as `AeadFailed` , not as distinct observable events. The oracle-indistinguishability argument from the decompression failure collapsing note in Theorem 13 (Property 1) applies here for the same reason — revealing whether a ciphertext decoded to wrong-size vs. structurally-invalid compressed data would create a 1-bit oracle beyond AEAD failure. A formal model for §9.11(c) must unify both post-AEAD collapse points under a single AeadFailed event; a model that introduces a distinct "decompressed-but-wrong-size" error branch introduces a spurious second distinguisher. This check also applies to both decryption paths (shared inner helper).

Per-chunk plaintext delivery: Unlike monolithic AEAD (which withholds all plaintext until the entire ciphertext is authenticated), this construction delivers each chunk's plaintext to the caller immediately upon successful authentication of that chunk — before the stream is complete. If authentication fails at chunk k, chunks 0..k-1 have already been delivered to the caller. The all-or-nothing authentication property holds only per-chunk, not per-stream. This does not constitute Release of Unverified Plaintext (RUP) in the [ADYM14] sense — each delivered chunk has been independently authenticated by a passing AEAD verification before delivery; no unverified plaintext is ever released. The [ADYM14] RUP concern applies to constructions that output plaintext before any MAC check; here each chunk's AEAD tag is verified before its plaintext is returned, satisfying per-chunk INT-CTXT. The composition concern is orthogonal: downstream components receive a stream of individually authenticated chunks and must handle the stream being incomplete if a later chunk fails. A formal composition proof must model the downstream component as receiving individual authenticated chunks rather than a single complete authenticated plaintext.

A formal model must represent these as distinct rules with the following state partition:

- **Linear** (modified by sequential rules only): next_index, finalized
- **Persistent** (set at init, read by both rules): key, base_nonce, version, flags, caller_aad

`next_index` is publicly observable via the `expected_index()` accessor on `StreamDecryptor` — it exposes the count of chunks successfully decrypted so far. This is traffic metadata (Channel 2, §8.5): an observer who can query the decryptor learns how many chunks have been processed without seeing the plaintext. A formal model that treats `next_index` as secret overspecifies the construction; models for Channel 2 analysis should treat it as an adversary-visible value.

The sequential rule consumes and re-creates a linear state fact (advancing next_index and possibly setting finalized); the random-access rule reads only the persistent fields without consuming state. **Independent encryptor/decryptor state**: A formal model must instantiate separate linear state facts for the encryptor and each decryptor — one `StreamEncState(next_index_enc, finalized_enc)` fact for the encryption oracle and one `StreamDecState(next_index_dec, finalized_dec)` fact for the sequential decryption oracle — initialized independently at (0, false) and sharing only the persistent fields (key, base_nonce, version, flags, caller_aad). The encryptor and decryptor are separate stateful objects: advancing the decryption oracle's next_index_dec does not affect next_index_enc, and vice versa. A model that shares a single linear state between the two oracles would incorrectly require that adversary sequential decryption queries advance the encryptor's counter, which is not the construction's behavior.

15.4 AAD Construction

Each chunk's AAD binds the chunk to its stream, position, finality, and application context:

```
aad = "lo-stream-v1"     (12 bytes, domain label)
   || version             (1 byte)
   || flags               (1 byte)
   || base_nonce           (24 bytes)
   || chunk_index          (8 bytes, big-endian u64)
   || tag_byte             (1 byte)
   || caller_aad           (variable, caller-supplied)
```

Total AAD size: 47 + len(caller_aad) bytes. caller_aad is the terminal field with no length prefix — the first 47 bytes have a fixed layout, so distinct caller_aad values always produce distinct AAD byte strings.

Security properties of the AAD fields:

Field	Prevents	Mechanism
`"lo-stream-v1"`	Cross-component confusion	Domain separation (Theorem 7)
`version`	Version downgrade	Changing version byte causes AEAD failure
`flags`	Flag flipping (e.g., compression bit)	Flipping flags changes AAD, causing AEAD failure
`chunk_index`	Reordering	Nonce binding (§15.2 mask positions 0–7, big-endian u64) is the primary mechanism for Property 3: distinct chunk indices produce distinct nonces, so AEAD verification fails at any mismatched sequential position. AAD binding (this field) provides independent defense-in-depth but is not required for the Property 3 reduction (see Property 3 note: "nonce mismatch alone causes AEAD failure"). A formal proof of Property 3 requires §15.2 nonce injectivity (sub-target (a) in §9.11) — the canonical reduction goes through nonce mismatch alone (nonce injectivity ⇒ AEAD failure at any mismatched sequential position). AAD chunk_index binding is defense-in-depth only: it provides a second independent argument but is not a load-bearing reduction path for Property 3. §9.11 sub-target (e) (AAD injectivity) is a defense-in-depth verification target, not a substitute for sub-target (a). A formal modeler should not treat §9.11(e) as the primary Property 3 proof obligation or interpret "AAD binding is present" as sufficient to establish Property 3 independently of the nonce argument.
`base_nonce`	Cross-stream splicing	Both nonce binding (§15.2, distinct base_nonces produce distinct per-chunk nonces at every same-position (chunk_index, tag_byte)) and AAD binding (this field) prevent cross-stream splicing. The AAD binding is unconditionally load-bearing for the formal INT-CTXT reduction: since base_nonce is a field of every chunk's AAD (§15.4), base_nonce_A ≠ base_nonce_B guarantees AAD_A ≠ AAD_B at every position regardless of nonce comparison, making any cross-stream ciphertext an unconditional INT-CTXT forgery. The nonce binding handles same-position attacks directly (different base_nonces XOR the same mask to produce different nonces) but does not unconditionally handle different-position attacks where nonces could coincide by birthday collision — that case requires the AAD argument. A formal proof of Property 5 should use the AAD argument as the primary reduction; see Property 5 same-key INT-CTXT reduction for the full two-case structure.
`tag_byte`	Finality stripping	Demoting a final chunk changes both nonce and AAD (defense-in-depth: either binding alone is sufficient — nonce binding via §15.2 mask position 8, or AAD binding via this field — but both are applied)
`caller_aad`	Context confusion	A file encrypted for context X fails AEAD in context Y

---

9. Open Research Problems

The following problems are scoped for individual verification efforts (course project, Master's thesis, or focused formal methods engagement). Each maps to specific sections above and to the theorems in §8.6.

9.1 LO-KEX Authentication and Secrecy

Model LO-KEX (§4) in Tamarin or ProVerif under the corruption model (§8.1-8.2). Prove or falsify Theorems 1-2. Starting points: §4.3-4.4 protocol steps, §8.3 LO-KEX freshness predicate, §8.4 assumptions. Of particular interest: formally verifying that σ_SI (Theorem 2b) provides the claimed proof-of-possession guarantee under the HybridSig EUF-CMA assumption, and that the combined HKDF info commitment plus signature binding prevents all impersonation attacks representable in the model. Note: because fp_IK_B = H(pk_IK_B) is embedded in SI and covered by σ_SI (§4.3 Step 4, Theorem 2(a) explicit branch), recipient binding can be verified from the signature rule alone — no separate KEM decapsulability lemma is required for this property, simplifying the Tamarin/ProVerif model.

Channel 2 note for §9.1 models: A `SessionInit` that is structurally rejected (wrong crypto version, malformed content) produces an observable response distinguishable from silence — the responder emits a rejection event rather than no event. This is a Channel 2 probe surface: an adversary can determine whether a party is online and what crypto version it accepts by sending crafted session inits and observing response presence. A Tamarin model of LO-KEX should emit an `Out(Reject)` fact on the rejection path to make this observable — models that silently drop invalid session inits misrepresent the Channel 2 behavior. The Channel 1 theorems (Theorems 1-2) are unaffected; this is noted here to prevent modelers from inadvertently hiding the probe surface in the formalization.

9.2 PCS Recovery Latency in the KEM Ratchet

Formalize the KEM ratchet (§5.2-5.4) and analyze break-in recovery under different corruption schedules. Verify Theorem 5 and characterize the boundary:

- (a) Single Corrupt(RatchetState, P, t_c) followed by clean ratchet steps — verify recovery at t_r.
- (b) Corrupt(RNG, encapsulator, t_r) during recovery — show recovery fails (expected, per F2).
- (c) Compound corruption (state + RNG at different times) — characterize the minimal conditions for recovery.

Starting points: §8.3 F1-F4, §7.2 design note on bidirectional KEM. This problem directly addresses the single-sided freshness trade-off that distinguishes KEM

ratchets from DH ratchets.

Worked trace (reference for modelers — illustrates why two direction changes are required):

```
t₀: Corrupt(RatchetState, Alice, t₀)
    Adversary learns: rk, ek_s_A, ek_r_A, sk_s_A, pk_s_A, pk_r_A

t₁: Bob sends (triggers KEM ratchet step §5.2):
    Encaps(pk_s_A) → (c_ratchet, ss)
    Adversary holds sk_s_A from t₀ → computes ss = Decaps(sk_s_A, c_ratchet)
    → Adversary derives (rk', ek_s_B) = KDF_Root(rk, ss)
    ⇒ NON-RECOVERING: F3 violated (decapsulator's sk_s was compromised at t₀)

t₂: Alice receives Bob's message, sets pending = T

t₃: Alice sends (triggers KEM ratchet step §5.2):
    (pk_s_A', sk_s_A') ← KeyGen()  [fresh, unknown to adversary]
    Encaps(pk_r = Bob's pk_s_B) → (c_ratchet', ss')
    (rk'', ek_s_A') ← KDF_Root(rk', ss')
    Adversary holds rk' from t₁ but NOT ss'
      (recovering ss' requires sk_s_B, which was generated at t₁ by Bob)
    ⇒ RECOVERING iff: ¬Corrupt(RNG, Alice, t₃) [F2]
                  AND ¬Corrupt(RatchetState, Bob) during [t₁, t₃] [F3]
                  AND Bob's (pk_s_B, sk_s_B) generated without
                      Corrupt(RNG, Bob, t₁) [F4]
```

The first direction change (t_1) fails because the adversary holds sk_s_A from the compromise. The second direction change (t_3) succeeds because Alice generates a fresh keypair and encapsulates to Bob's post-compromise pk_s_B, introducing entropy the adversary cannot derive. This makes the F1-F4 conditions concrete.

Chain-injection extension (illustrates why F4 is necessary): After t_0, the adversary can inject NewEpoch messages with adversary-chosen keypairs (pk_s*, sk_s*) and known KEM ciphertexts. Each injected message that Alice accepts at §5.4 Step 3c sets pk_r ← pk_s* (adversary-controlled). Alice's next Send (§5.2) performs Encaps(pk_r = pk_s*) — the adversary holds sk_s* and can compute the KEM shared secret, extending the compromise through another epoch. This chain can repeat indefinitely: each injected epoch substitutes a new adversary-controlled pk_r, preventing recovery. F4 breaks this chain: recovery at t_r requires that the decapsulator's key pair encapsulated to at t_r was honestly generated — i.e., pk_s_B at t_r must belong to Bob, not the adversary. Without F4, the adversary could satisfy F2 (uncompromised encapsulator randomness at Alice's Send) while still computing ss (because they hold the decapsulation key sk_s* that Alice encapsulated to).

9.3 Domain Separation and Cross-Protocol Confusion

Verify Theorem 7: show that the label/AAD binding scheme prevents transcript reuse across protocol components. Enumerate all domain separation points (constants appendix in the implementation spec), model the AAD construction (§5.3-5.4 header encoding, §7.2 security properties), and show no adversarial relabeling produces a valid transcript in a different component. Sub-targets: (a) Encode disjointness — Encode(SI) and Encode(H) have disjoint length ranges (Theorem 7 sub-lemma, §8.6). (b) Info string injectivity — KDF_KEX's length-prefixed info construction and KDF_Call's fixed-size info construction are injective (§2.5); the KDF_KEX argument depends on the hard-fail version policy (§4.2 Step 3) for label-length unambiguity. (c) Nonce Uniqueness mechanism independence — the Nonce Uniqueness Lemma (§7.2) lists four mechanisms and claims each is necessary. A complete verification would produce four independent Tamarin lemmas, each removing one mechanism and exhibiting a nonce-reuse trace: (i) remove fork prevention → two copies encrypt with same (ek, counter); (ii) remove min_epoch integrity → storage replay rewinds counter; (iii) remove ChainExhausted guards → counter overflow wraps to 0; (iv) remove counter-0 retirement → session-init nonce collides with ratchet counter 0. This serves as a soundness check: if any mechanism is proven redundant, the lemma statement has an error. **Tool-dependent asymmetry**: of the four targets, (i) fork prevention and (ii) min_epoch integrity are directly testable in a symbolic model (removing the corresponding linear-fact or monotonicity encoding produces counterexample traces). (iii) ChainExhausted guards require a bounded counter model or an explicit overflow rule (symbolic successor terms do not overflow). (iv) Counter-0 retirement is only testable in a computational model (CryptoVerif / game-based proof): in a symbolic model, the session-init random nonce ~n0 is a fresh name that structurally cannot equal the counter-derived term $0^{20} \| \text{BE32}(0)$, so removing counter-0 retirement produces no nonce-collision trace — the target is vacuously true. A Tamarin modeler following (iv) literally would get a vacuously true result and might incorrectly conclude the mechanism is redundant.

9.4 Counter-Mode Duplicate Detection Bounds

Formalize the recv_seen state machine (§5.1 invariants a-i, §5.4 duplicate detection) and prove:

- (a) The recv_seen set never exceeds MAX_RECV_SEEN entries under any adversarial message schedule.
- (b) The prev_ek_r grace period correctly handles late-arriving messages from the immediately preceding epoch without key material leakage.
- (c) The set resets on each KEM ratchet step, preventing unbounded accumulation across epochs.

Amenable to a bounded model checker or direct inductive proof.

9.5 LO-Call Key Independence

Prove that call keys (§5.7) are independent of ratchet message keys derived from the same root key rk, under the HKDF dual-PRF assumption (§8.4). Show that Corrupt(CallKeys, P, call_id) does not violate LO-Ratchet message secrecy and vice versa. Starting points: §7.3 LO-Call security properties, §8.3 LO-Call freshness predicate. Note that call key derivation reads rk but does not modify Σ — the independence argument rests on the HKDF info-string separation between `"lo-call-v1"` and `"lo-ratchet-v1"`.

Concurrent call independence (Theorem 11): Extend to multiple concurrent calls sharing the same rk. Model two or more KDF_Call invocations with distinct call_id values under the same rk and verify joint independence of the resulting call keys. The reduction embeds the IND-CCA2 challenge at each ephemeral encapsulation (§5.7 Step 2); distinct IKM inputs ($ss_{eph_1} \| call\_id_1$ vs $ss_{eph_2} \| call\_id_2$) produce independent HKDF-Extract outputs. Show that Corrupt(CallKeys, P, $call\_id_1$) does not reveal keys for $call\_id_2 \neq call\_id_1$. The security loss is additive in the number of concurrent calls.

9.6 Counter-Mode vs Chain-Mode Key Derivation Security

LO-Ratchet derives message keys via `KDF_MsgKey(ek, counter) = MAC(key=ek, data=0x01 ‖ BE32(counter))` — all messages within an epoch share a single epoch key (§2.3). This trades per-message forward secrecy for O(1) out-of-order decryption without a skip cache. Formally characterize the security difference:

- (a) Under `Corrupt(RatchetState, P, t)` at time t within an epoch, the adversary recovers the epoch key and can derive all message keys (past and future) within that epoch. Quantify the exposure window compared to a chain-mode ratchet where only forward-derivable keys are exposed.
- (b) Show that the epoch key ek and root key rk reside in the same trust domain (same memory, same compromise granularity), confirming that per-message chain advancement provides no practical security benefit when the epoch key is co-resident with rk (the design rationale in §7.2).
- (c) Verify that the `recv_seen` duplicate-detection mechanism provides equivalent replay resistance to a chain-mode skip cache under the bounded-set invariant (|recv_seen| ≤ MAX_RECV_SEEN).

9.7 OPK Atomicity Under Concurrent Session Inits

Model the OPK consumption mechanism (§4.4 Step 6, A5 in §7.5) under concurrent session initiation. Two session-init messages referencing the same OPK id arrive concurrently:

- (a) Verify that at most one session succeeds — the one that atomically consumes the `!OPK(id, sk)` linear fact. The second must fail at Step 4 (decapsulation fails because sk_OPK has been deleted).
- (b) Verify that no execution trace exists where both sessions derive the same ss_OPK (which would violate the forward secrecy enhancement that OPK provides).
- (c) Characterize the behavior when atomicity is violated (TOCTOU window): both sessions derive identical ss_OPK, identical (rk, ek), and identical initial ratchet states — a session duplication that the application layer cannot distinguish from a legitimate session.

Amenable to a bounded Tamarin model with explicit concurrency (two parallel SessionInit rules competing for the same linear OPK fact). Course-project scale. Directly tests the A5 ruling from §7.5.

9.8 SPK Rotation Lifecycle

§3 states SPKs are "rotated approximately weekly; old secret keys retained for a grace period," but the adversary model (§8.2) treats Corrupt(SPK, P, id) as a point query without modeling the rotation lifecycle. Formalize the interaction between SPK rotation and LO-KEX security properties:

- (a) **Grace period semantics**: During rotation, both sk_SPK[id_old] and sk_SPK[id_new] exist simultaneously. The LO-KEX freshness predicate (§8.3) requires no corruption of sk_SPK_B[id_SPK] — but id_SPK is the one in the bundle that Alice fetched, which may be stale. Model the grace-period window as a set of concurrently valid SPK identifiers and verify that freshness holds for the id_SPK in Alice's bundle, not the "current" one.
- (b) **Deletion timing and forward secrecy**: LO-KEX forward secrecy (§7.1) requires sk_SPK_B[id_SPK] to have been deleted. The grace period delays this deletion, creating a window where KEX-level forward secrecy is not yet achieved for sessions established using that SPK. Characterize this window and its interaction with the ratchet's per-epoch forward secrecy (Theorem 4) — once the first KEM ratchet step completes, session secrecy no longer depends on sk_SPK.
- (c) **Stale bundle liveness**: If Alice caches a bundle with id_SPK = N and Bob has since rotated to id_SPK = N+1 and deleted sk_SPK[N], Alice's session init fails at §4.4 Step 5 (SPK lookup). This is a liveness issue, not a security one, but a formal model must handle the failure trace.

Course-project scale. Tests the interaction between key management and the LO-KEX security properties.

9.9 Anti-Reflection Verification

Verify Theorem 12: show that reflected messages (A→B replayed as B→A within the same session) fail AEAD verification due to reversed fingerprint ordering in the AAD. **Known modeling pitfall**: a Tamarin/ProVerif model that uses canonical (e.g., lexicographically sorted) fingerprint ordering in the AAD construction rule will not detect reflection attacks — the AAD would be identical in both directions, and the model would produce false counterexamples. The AAD rule must preserve the sender-first ordering: `"lo-dm-v1" ‖ fp_sender ‖ fp_recipient ‖ Encode(H)` . Starting points: §7.2 anti-reflection paragraph, invariant (h) (local_fp ≠ remote_fp). The proof is a one-step reduction to INT-CTXT: the reflected ciphertext's reconstructed AAD differs from the original, so the AEAD tag is invalid. Amenable to automated verification; the main value is as a sanity check that the model's AAD encoding is correct.

Note on Theorem 6 (LO-Auth): Theorem 6 is intentionally omitted as a standalone research problem. The proof is a standard IND-CCA2 + PRF reduction: the adversary's forgery advantage reduces to distinguishing Decaps output from random (IND-CCA2 on X-Wing) and then distinguishing the HMAC tag from random (PRF on HMAC-SHA3-256). The compositional interaction with Theorem 1 (LO-Auth sessions as CCA2 oracles on sk_IK[XWing]) is already documented in the Theorem 1 compositional note and the §8.3 AuthPending linear fact lifecycle. No structural subtleties remain beyond what is stated.

9.11 Streaming AEAD Construction Verification

Verify Theorem 13: show that the chunked AEAD construction (§15) provides the claimed confidentiality, integrity, ordering, truncation resistance, and cross-stream isolation properties. The construction is standard (deterministic nonce derivation over XChaCha20-Poly1305), so the primary value is as a sanity check that the AAD/nonce binding is correct — parallel to §9.9 (Anti-Reflection). Sub-targets:

- (a) **Nonce injectivity**: Verify the §15.2 injectivity claim under the full (index, tag_byte, base_nonce) domain. For the multi-stream case, verify the $(\Sigma M_i)^2/2^{193}$ birthday bound on cross-stream nonce collisions over total AEAD invocations (Theorem 13, multi-stream note). Note: the bound is over total chunks, not total streams — a per-chunk collision (specific base_nonce difference matching a single mask-XOR target) is less catastrophic than a base_nonce collision (all positions collide) but still constitutes an IND-CPA break. **Tool applicability**: The multi-stream birthday bound is a probabilistic argument. In a symbolic model (Tamarin/ProVerif), base_nonce ← Fr produces structurally distinct names — cross-stream nonce collision is unreachable by construction, making the birthday bound vacuously satisfied. A modeler targeting multi-stream nonce uniqueness must either use a computational model (CryptoVerif) or introduce an explicit collision axiom. This parallels §9.3(c)(iv), where counter-0 retirement is vacuously true in a symbolic model.
- (b) **Sequential vs random-access security delta**: Formalize the properties that the sequential interface provides (ordering via implicit counter, truncation detection via finalized flag) and show that the random-access interface provides neither. This should produce two distinct sets of Tamarin lemmas: ordering/truncation lemmas that succeed for sequential rules and produce counterexamples for random-access rules.
- (c) **Decompression oracle indistinguishability**: Verify that the post-AEAD decompression failure collapse (Theorem 13 property 1 note) produces no observable distinction from AEAD failure in the implemented error model. A model that separates these failure events should produce a distinguishing trace; collapsing them should eliminate it. **Note on scope**: This is a model fidelity obligation, not a cryptographic proof obligation. Under INT-CTXT with a fresh key, honest ciphertexts (∈ L, produced by encrypting ≤CHUNK_SIZE plaintexts with the reference compressor) always decompress successfully — the 1-bit oracle that would distinguish "valid AEAD, bad compression" from "failed AEAD" is unreachable in any fresh-key game, because non-L ciphertexts fail AEAD authentication before any decompression attempt. §9.11(c) therefore verifies that the formal model does not introduce a spurious reachable branch that the real protocol does not have, rather than providing an independent cryptographic guarantee beyond INT-CTXT.
- (d) **`encrypt_chunk_at` precondition tightness**: Verify that violating the caller-enforced distinct-(index, tag_byte)-pair constraint on `encrypt_chunk_at` (§15.3) produces an immediate distinguisher. A formal model that permits repeated (index, tag_byte) pairs under the parallel interface should yield a nonce-

reuse counterexample trace (the same (key, nonce) pair encrypts two different plaintexts, breaking IND-CPA). Enforcing the precondition as a game guard should eliminate the trace. This establishes that the distinct-pair contract is the precise IND-CPA boundary for this interface — a cryptographic necessity, not a usability constraint.

- (e) **AAD injectivity**: Verify that the §15.4 AAD construction is injective over (version, flags, base_nonce, chunk_index, tag_byte, caller_aad).
- (f) **Q-parameterized security bound**: In CryptoVerif, formalize both the IND-CPA and INT-CTXT reductions as functions of the session parameter Q (and the Q_seq/Q_par split when `encrypt_chunk_at` is co-exposed). Verify mechanically that: (i) the IND-CPA proof incurs a Q-step hybrid where each step contributes Adv_IND-CPA(λ) — concrete bound Q × Adv_IND-CPA(XChaCha20-Poly1305); (ii) the INT-CTXT reduction is a single forgery-forwarding step with no Q-factor — concrete bound Adv_INT-CTXT(XChaCha20-Poly1305) regardless of Q. This sub-target provides a formal basis for the concrete safe operating limit (Q ≤ 2^{32} chunks for ≥96-bit IND-CPA, per the Security loss note in Property 1). Without a mechanized Q-parameterized proof, the specific threshold is a manual calculation that a reviewer must accept on faith. The IND-CPA/INT-CTXT divergence in Q-dependence — where both notions are provided by the same AEAD but at different Q-scaled losses — is the key non-obvious property that this sub-target validates, and is not directly analogous to any monolithic AEAD usage pattern in the prior verification literature. The 47-byte fixed-layout prefix (§15.4: 12-byte label (`"lo-stream-v1"`) + 1-byte version + 1-byte flags + 24-byte base_nonce + 8-byte chunk_index + 1-byte tag_byte + variable caller_aad) produces distinct byte strings for distinct (base_nonce, chunk_index, tag_byte) triples — a structural fixed-length parsing argument. This lemma is distinct from nonce injectivity (sub-target (a)), which uses an XOR-bijection argument over §15.2; the AAD injectivity uses a layout-based argument over §15.4. Properties 3 and 4 both reduce to "INT-CTXT under the injective nonce/AAD construction (§15.2, §15.4)" — a Tamarin model represents the nonce and the AAD as structurally distinct constructor terms, and the injectivity of each requires an independent proof.

Course-project scale. Amenable to automated verification; the nonce injectivity sub-target (a) is a straightforward algebraic argument, while (b) and (c) require modeling the streaming state machine. **Proof-structure note**: The LO construction replaces [HRRV15]'s ciphertext chaining with AAD binding of chunk_index, so the [HRRV15] OAE2 proof does not apply (the random-access interface explicitly breaks the sequential dependency that OAE2 requires). Beyond proof inapplicability, the construction does not satisfy OAE2: OAE2 requires online AE that remains secure even under base_nonce reuse (misuse-resistant online AE), while LO requires nonce uniqueness. A base_nonce reuse under the same key forces identical per-chunk nonces at equal (chunk_index, tag_byte) positions (chunk_nonce = base_nonce XOR mask; same base_nonce → same mask → same nonce), so XOR of the two streams' ciphertexts directly reveals XOR of plaintexts — a trivial IND-CPA break. Since OAE2 implies MRAE in the online setting, the NMR disclaimer (Theorem 13 oracle scope note, [RS06]) implies OAE2 non-satisfaction as a corollary. A formal reviewer should not attempt to verify OAE2 as a Theorem 13 corollary. **OAE1 is equally inapplicable as a proof path** : [HRRV15] defines both OAE2 (misuse-resistant, discussed above) and OAE1 (nonce-based online AE, nonce uniqueness required). The sequential sub-game's property set — {confidentiality, integrity, ordering, truncation detection} — coincides with OAE1's scope, so a reviewer might ask whether OAE1 can be cited instead of OAE2. It cannot: [HRRV15]'s OAE1 proof uses the same ciphertext-chaining structure as OAE2 for its ordering and truncation arguments — each segment's authentication tag feeds into the next segment's nonce derivation, making reordering and truncation detectable without any explicit counter in the AAD. Since the LO construction replaces that chain with per-chunk §15.2 nonce injectivity and §15.4 AAD binding (no inter-chunk dependency), neither OAE variant's proof path carries over. The ordering and truncation arguments here follow [BKN04]'s stAE reduction (counter-derived nonces and stateful decryptor) rather than [HRRV15]'s chaining reduction. The correct reduction goes directly from AEAD IND-CPA + INT-CTXT under injective nonces (§15.2) and AAD binding (§15.4), without inter-chunk dependency in the base case. The dual-oracle game (sequential + random-access on the same stream) reduces cleanly because the random-access oracle reads only persistent state facts and cannot interfere with the sequential oracle's linear state (§15.3 state partition) — each oracle reduces independently to per-chunk AEAD queries. Ordering (property 3) is then a separate argument from the AAD's chunk_index field, not from ciphertext chaining. **Random-access security profile**: A caller using only `decrypt_chunk_at` (no sequential oracle) has a well-defined security subset: {Property 1 (IND-CPA), Property 2 (INT-CTXT, both interfaces), Property 5 (cross-stream isolation)}. Properties 3 (ordering) and 4 (truncation resistance) are inapplicable — the random-access interface by design accepts any valid (chunk_index, tag_byte, ciphertext) without enforcing ordering or finalization. A formal model for a random-access use case (parallel downloader, indexed file system) should scope its verification obligations to this three-property subset and omit the sequential-oracle lemmas entirely. This three-property subset — per-chunk IND-CPA + INT-CTXT + cross-stream isolation for an index-addressed AEAD — has no direct named precedent in the AE literature (the closest prior notions, OAE2 [HRRV15] and stAE [BKN04], both include ordering/completeness in their definitions); it is a novel construction without prior-art security reduction that can be cited directly. **Mixed-interface security profile**: The most common practical deployment pattern — sequential encryption (`encrypt_chunk`) on the sender side, random-access decryption (`decrypt_chunk_at`) on the receiver side (e.g., server encrypts a file sequentially; clients download and decrypt chunks by index) — has a distinct security profile from either the pure-sequential or pure-random-access cases. On the decryption side, the security subset is {Property 1, Property 2, Property 5}: the same three properties as the pure-random-access profile, since `decrypt_chunk_at` provides neither ordering nor truncation detection regardless of how the ciphertext was produced. On the encryption side, the sequential encryptor's `is_finalized()` flag is available, but it reflects the *sender's* stream completion — it does not propagate to the random-access decryptors, who have no equivalent finalization state. Property 4 (truncation resistance) is a caller obligation for mixed-interface receivers: the caller must externally track the set of received chunk indices {0..N}, identify the final-chunk index from the is_last return value of `decrypt_chunk_at` , and verify that the complete set {0..N} has been received before treating the stream as complete. A formal model for this pattern should include the sequential encryption oracle, a random-access decryption oracle (no `finalized` state), and an explicit out-of-band completeness check as a caller-side obligation rather than a protocol guarantee. **Tamarin two-party state partition**: Represent three distinct fact classes — (1) shared persistent fact `!StreamPersistentState(stream_id, key, base_nonce, version, flags, caller_aad)` set at stream setup, read by all rules on both parties; (2) sender-side linear fact `StreamEncState(stream_id, next_index_enc, finalized_enc)` consumed and re-produced by the sequential EncryptChunk rule, absent from all receiver rules — receiver rules must not reference or consume this fact; (3) no linear state fact for receivers (stateless random-access). The sender's `is_finalized()` flag lives only in `StreamEncState` — no protocol message carries it to receivers, so a Tamarin receiver rule must not include `finalized_enc` in any premise. The out-of-band completeness check on the receiver side is a separate Tamarin rule that fires when the receiver has independently collected the complete index set {0..N} and observed the is_last flag from `decrypt_chunk_at(N)` — its conclusion (e.g., `ReceiverStreamComplete(receiver, stream_id)`) is the fact that receiver-side security lemmas should use as a premise for stream-level guarantees. **Adversary oracle structure**: The mixed-interface adversary has access to (i) the sequential EncryptChunk oracle (submits plaintexts, receives ciphertexts on the sender's stream), (ii) DecryptChunkAt oracle (submits (stream_id, index, tag_byte, ciphertext) to any receiver), and (iii) Corrupt(StreamKey). Properties 1, 2, and 5 are stated as in the random-access security profile; the mixed-interface adversary has strictly fewer capabilities than the full dual-oracle adversary for Properties 3 and 4 (no sequential decryption oracle), so Properties 3 and 4 are not adversary goals in this configuration — they are receiver caller obligations.

9.10 Full-Protocol Composition Under Compound Corruption

The spec proves individual theorems (1-12) and provides pairwise composition results: KEX→Ratchet (§8.6 Composition paragraph) and Ratchet→Call (Theorem 10). The transitive composition KEX→Call is implicit: call key secrecy (Theorem 8) depends on rk freshness, which depends on KEX session key freshness (Theorem 1), but this two-hop reduction chain is not explicitly stated.

A full composition theorem would establish: Theorems 1, 3-5, and 8-12 hold simultaneously under the full adversary model (§8.1-8.2) with compound corruption schedules that span all three protocol layers. Specifically:

- (a) **KEX→Ratchet→Call transitivity**: A single compound corruption trace (e.g., Corrupt(IK) before KEX, then Corrupt(RNG) during a ratchet step, then Corrupt(CallKeys) during a call) — verify that the individual theorem conclusions compose without interference. The security loss should be additive across the three layers.
- (b) **Cross-layer oracle interactions**: LO-Auth sessions provide Decaps oracle access on sk_IK[XWing] (Theorem 1 compositional note); LO-Call ephemeral encapsulations provide additional KEM queries. Verify that the total number of oracle queries across all protocol components remains within the IND-CCA2 reduction's budget.
- (c) **Shared-state dependencies**: rk is read by both KDF_Root (ratchet advancement) and KDF_Call (call derivation). The dual-use salt note on Theorem 10 provides the pairwise argument; extend to the case where KEX-derived rk is simultaneously used for the initial ratchet epoch and a call initiated before the first KEM ratchet step.

This is a larger verification effort than §9.1-9.11 (likely a focused formal methods engagement or thesis chapter). The individual pairwise results provide a foundation; the remaining work is showing that compound corruption schedules do not create interference between the pairwise arguments.

Streaming layer (Theorem 13): At the cryptographic level, the streaming AEAD layer composes cleanly with the rest of the protocol — stream keys are caller-provided and not derived from protocol state (§15.1), so `Corrupt(StreamKey)` and `Corrupt(RatchetState)` are independent by construction, and a composition argument need only verify that the domain separation label `"lo-stream-v1"` (Theorem 7) prevents cross-component confusion (already a sub-target of §9.3). However, this clean independence holds for the abstract standalone streaming layer where key delivery is not modeled. In the primary usage pattern — where the ratchet delivers a stream key k as application-layer message content at send position i — the composition is not trivial: `Corrupt(MessageKey, P, i)` at delivery position i reveals mk, which decrypts the delivery message to reveal k, transitively compromising all streams under k without any `Corrupt(StreamKey)` or `Corrupt(RatchetState)` query being issued. This indirect corruption path is not captured by the per-party stream freshness predicate or the standard F1–F4 ratchet predicate. A full-composition model of this §9.10 type must include the conditions documented in Theorem 13's Composed-protocol freshness paragraph and §15.1 (Indirect corruption path and Cross-party key sharing notes), or the resulting composition theorem will be unsound for the ratchet-delivered-key case.

10. References

- [ACD19] Alwen, J., Coretti, S., Dodis, Y. "The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol." EUROCRYPT 2019.
- [CGCD+20] Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D. "A Formal Security Analysis of the Signal Messaging Protocol." Journal of Cryptology, 2020. (Conference version: EuroS&P 2017.)
- [HKS+22] Hashimoto, K., Katsumata, S., Kwiatkowski, K., Prest, T. "An Efficient and Generic Construction for Signal's Handshake (X3DH): Post-Quantum, State Leakage Secure, and Deniable." Journal of Cryptology, 2022. (Conference version: PKC 2021.)
- [BFG+20] Brendel, J., Fischlin, M., Günther, F., Janson, C., Stebila, D. "Towards Post-Quantum Security for Signal's X3DH Handshake." SAC 2020. https://eprint.iacr.org/2019/1356
- [XWing] Connolly, D., Hülsing, A., Joseph, D., Liu, F.-H., Schmieg, J., Schwabe, P. "X-Wing: The Hybrid KEM You've Been Looking For." draft-connolly-cfrg-xwing-kem-09.
- [FIPS203] NIST. "Module-Lattice-Based Key-Encapsulation Mechanism Standard." FIPS 203, 2024.
- [FIPS204] NIST. "Module-Lattice-Based Digital Signature Standard." FIPS 204, 2024.
- [RFC8032] Josefsson, S., Liusvaara, I. "Edwards-Curve Digital Signature Algorithm (EdDSA)." RFC 8032, 2017.
- [RFC5869] Krawczyk, H., Eronen, P. "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)." RFC 5869, 2010.
- [RFC7748] Langley, A., Hamburg, M., Turner, S. "Elliptic Curves for Security." RFC 7748, 2016.
- [HRRV15] Hoang, V.T., Reyhanitabar, R., Rogaway, P., Vizár, D. "Online Authenticated-Encryption and its Nonce-Reuse Misuse-Resistance." CRYPTO 2015.
- [BN00] Bellare, M., Namprempre, C. "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm." ASIACRYPT 2000. IACR ePrint 2000/025.
- [BKN04] Bellare, M., Kohno, T., Namprempre, C. "Breaking and Provably Repairing the SSH Authenticated Encryption Scheme: A Case Study of the Encode-then-Encrypt-and-MAC Paradigm." ACM Transactions on Information and System Security, 2004. (Conference version: CCS 2002.)
- [Rog04] Rogaway, P. "Nonce-Based Symmetric Encryption." Fast Software Encryption (FSE) 2004. Introduces the nAE (nonce-based AE) model in which nonces are adversary-observable and security requires only nonce uniqueness, not nonce secrecy.
- [RS06] Rogaway, P., Shrimpton, T. "A Provable-Security Treatment of the Key-Wrap Problem." EUROCRYPT 2006. Introduces DAE (Deterministic Authenticated Encryption) and the SIV construction. The related concept of MRAE (Misuse-Resistant AE) — nonce-based AE that degrades gracefully to DAE security under nonce reuse — builds on this foundation. The LO streaming construction operates in the nAE model [Rog04] and does not achieve DAE or MRAE.
- [Bel06] Bellare, M. "New Proofs for NMAC and HMAC: Security without Collision-Resistance." CRYPTO 2006. Establishes PRF and dual-PRF properties of HMAC under compression function assumptions without requiring Merkle-Damgård structure, covering arbitrary underlying hash functions.
- [Krawczyk10] Krawczyk, H. "Cryptographic Extraction and Key Derivation: The HKDF Scheme." CRYPTO 2010. IACR ePrint 2010/264. Proves that HKDF-Extract is a randomness extractor (dual-PRF property of HMAC) and that HKDF-Expand is a PRF keyed by the extracted PRK. Load-bearing for §8.4's dual-PRF assumption, which underlies KDF_Root / KDF_KEX security and transitively Theorems 3, 4, 5, 10, and 11. Note: the analysis targets HMAC-SHA2; applicability to HMAC-SHA3-256 follows via the generic HMAC construction [Bel06], as noted in §8.4.
- [BT16] Bellare, M., Tackmann, B. "The Multi-User Security of Authenticated Encryption: AES-GCM in TLS 1.3." CRYPTO 2016. Defines the MI-AEAD security framework bounding joint adversarial advantage across N concurrent encryption instances; the per-stream security of Theorem 13 implies MI security with advantage additive across streams plus birthday overhead.
- [ADYM14] Andreeva, E., Bogdanov, A., Luykx, A., Mennink, B., Mouha, N., Yasuda, K. "How to Securely Release Unverified Plaintext in Authenticated Encryption." ASIACRYPT 2014. Defines the RUP (Release of Unverified Plaintext) security notion; the per-chunk INT-CTXT guarantee of Theorem 13 means each chunk's plaintext is delivered after verification, satisfying per-chunk authenticated encryption and not triggering RUP concerns.