

# SiloSec Protocol Specification

## The Definitive Blueprint for Silo-to-Silo Encrypted Communication

**Registry:** [[HOWL-COMP-8-2026](#)]

**Series Path:** [[HOWL-COMP-1-2026](#)] → [[HOWL-COMP-2-2026](#)] → [[HOWL-COMP-3-2026](#)] → [[HOWL-COMP-4-2026](#)] → [[HOWL-COMP-5-2026](#)] → [[HOWL-COMP-6-2026](#)] → [[HOWL-COMP-7-2026](#)] → [[HOWL-COMP-8-2026](#)]

**Parent Framework:** [[HOWL-COMP-1-2026](#)]

**DOI:** 10.5281/zenodo.zzz

**Date:** February 2026

**Domain:** Software Architecture / Systems Engineering / Real-Time Computing

**Status:** Architectural Blueprint for Independent Implementation

**AI Usage Disclosure:** Only the top metadata, figures, refs and final copyright sections were edited by the author. All paper content was LLM-generated using Anthropic's Claude 4.5 Sonnet.

---

## Table of Contents

1. Philosophy & Motivation
  2. [What We Rejected and Why](#)
  3. [Protocol Design](#)
  4. [Cryptographic Primitives](#)
  5. [Wire Format](#)
  6. [Handshake Protocol](#)
  7. [Data Transmission](#)
  8. [Implementation Guide](#)
  9. [Integration with Silo OS](#)
  10. [Security Considerations](#)
  11. [Future Claude Integration Notes](#)
-

## Philosophy & Motivation

### The Problem

Traditional secure communication protocols (TLS, IPSec) are built on decades of accumulated complexity:

- **TLS 1.3:** 15,000+ lines of code, requires RSA/X.509/ASN.1/PKI infrastructure
- **IPSec:** 8,000+ lines, kernel integration nightmares, debugging impossibility
- **Both:** Depend on certificate authorities, expiring certificates, cipher negotiation, version management

### Quote from design discussion:

“we abandon wifi, in cases its required, use a VM and pretend its cabled”

“this is an OS, we are restarting computing, some crypto is needed.”

### The Realization

**Key insight:** In a 1-process OS where the kernel IS the application, there is no separation between “kernel” and “userspace” networking. SiloSec can operate at the IP layer without “kernel integration” because it IS the kernel.

**Second insight:** We only need to encrypt Silo-to-Silo communication. Legacy machines get plain HTTP or nothing.

**Third insight:** Zig’s standard library already contains all necessary primitives:

```
std.crypto.dh.X25519                // Key exchange
std.crypto.sign.Ed25519             // Authentication
std.crypto.aead.aes_gcm.Aes256Gcm  // Encryption
No external dependencies. No C code. No binary blobs.
```

### Design Constraints (The K Split)

From the original philosophy:

“Low road chosen - \$200 CPU, pure software, no drivers, runs anywhere, accessible to all.”

Applied to networking:

- **NO** proprietary firmware (WiFi rejected)
- **NO** external crypto libraries (BearSSL rejected, even at 50KB)

- **NO** complexity we can't audit (TLS/IPSec rejected)
- **YES** to minimal, auditable, pure Zig code

**Quote from BearSSL rejection:**

“i thought 50kb of C, rejecting, too big. not worth it.”

This is correct. 15,000 lines of C we cannot audit violates everything Silo stands for.

---

## **What We Rejected and Why**

### **Rejected: HTTPS/TLS**

#### **Complexity:**

- TLS 1.2/1.3 implementation: ~15,000 lines
- X.509 certificate parsing: ~2,000 lines
- ASN.1 DER encoding: ~1,500 lines
- RSA operations: ~10,000 lines (big integer math)
- **Total:** ~28,500 lines minimum

#### **Even with BearSSL:**

- 50KB compiled C code
- 15,000 lines we cannot audit
- Binary blob we don't control
- Defeats minimal, auditable OS principle

#### **Why rejected:**

“50kb of C, rejecting, too big. not worth it.”

TLS requires trust in:

- Certificate authorities
- Browser vendors
- Certificate revocation infrastructure
- Cipher suite negotiation
- Version compatibility

**We reject all of this as “dead wood.”**

### **Rejected: IPSec**

#### **Quote from analysis:**

“doesn't require kernel integration in a 1 process OS”

**Why we considered it:** IPSec operates at IP layer, seemed like natural fit.

#### **Why we rejected it:**

1. **Complexity:** Still ~8,000 lines minimum
2. **IKE Protocol:** Requires same RSA/X.509 infrastructure as TLS
3. **All-or-nothing:** Encrypts everything or nothing (no per-peer control)
4. **Legacy baggage:** Designed for traditional OS architecture
5. **Debugging nightmare:** Even worse than TLS

**Conclusion:** IPSec is NOT simpler than TLS. It's complexity dressed in different clothes.

### **Rejected: SSH Tunnels / Proxies**

#### **Quote from discussion:**

“there is no SSH, not an option”

“this is an OS, we are restarting computing”

SSH tunnels and proxies assume:

- Another OS exists to run the proxy
- We're operating as userspace on legacy system
- We can delegate complexity elsewhere

**This violates the premise.** Silo OS IS computing. There is no “host OS” to tunnel through.

### **Rejected: Application-Layer Encryption Only**

Initially considered:

```
// Just encrypt at application layer  
  
const encrypted = encryptAES_GCM(data, key);  
  
http.post("http://server.com", encrypted);
```

#### **Problems:**

1. Leaves key exchange unsolved (how do peers agree on key?)
2. No authentication (anyone can claim to be anyone)
3. Vulnerable to replay attacks
4. No forward secrecy
5. Requires manual key management

**Conclusion:** Application-layer crypto alone is insufficient for a secure networking primitive.

---

## Protocol Design

### Core Principles

1. **Frozen Protocol:** Version 1, never changes. No negotiation, no upgrades.
2. **Identity = Public Key:** No certificates, no PKI, no expiration.
3. **Trust On First Use (TOFU):** First contact establishes identity.
4. **One Cipher Suite:** AES-256-GCM. Period. No options.
5. **IP Layer:** Works below TCP/UDP, encrypts all traffic to peer.
6. **Silo-Only:** Only Silo machines speak SiloSec. Legacy gets plaintext or rejection.

### Design Rationale

#### Why IP layer (not transport layer)?

- Encrypts ALL traffic (TCP, UDP, ICMP, custom protocols)
- Single handshake per peer (not per connection)
- In 1-process OS, IP layer IS application layer
- Simpler than per-connection TLS

#### Why X25519/Ed25519?

- Modern, secure, fast
- Constant-time implementations (timing attack resistant)
- Small key sizes (32 bytes)
- Already in Zig stdlib
- Proven security properties

#### Why no cipher negotiation?

“assume all old crypto is dead wood”

If AES-256-GCM is broken, we're all screwed anyway. Cipher agility adds complexity for zero security benefit.

### **Why no versioning?**

Protocol is frozen at v1. If we need changes:

1. Fork the protocol (SiloSec2)
2. Run both in parallel
3. Eventually deprecate v1

This is simpler than in-band version negotiation.

---

## **Cryptographic Primitives**

All primitives from Zig stdlib (`std.crypto`):

### **X25519 (Curve25519 Diffie-Hellman)**

**Purpose:** Ephemeral key exchange for forward secrecy

**Properties:**

- 128-bit security level
- Constant-time operations
- 32-byte keys
- Fast (~50,000 operations/sec on \$200 CPU)

**Usage:**

```
// Alice generates ephemeral keypair
var alice_secret: [32]u8 = random_bytes();

var alice_public = crypto.dh.X25519.scalarMultBase(alice_secret);

// Bob generates ephemeral keypair
var bob_secret: [32]u8 = random_bytes();

var bob_public = crypto.dh.X25519.scalarMultBase(bob_secret);

// Both compute same shared secret
var shared_secret_alice = crypto.dh.X25519.scalarMult(alice_secret, bob_public);
```

```
var shared_secret_bob = crypto.dh.X25519.scalarMult(bob_secret, alice_public_key);  
// shared_secret_alice == shared_secret_bob
```

### **Ed25519 (Edwards-curve Digital Signature)**

**Purpose:** Peer authentication and identity

**Properties:**

- 128-bit security level
- Constant-time operations
- 32-byte public keys, 64-byte signatures
- Deterministic (same message always produces same signature)

**Usage:**

```
// Generate identity keypair  
var seed: [32]u8 = random_bytes();  
var keypair = crypto.sign.Ed25519.KeyPair.create(seed);  
  
// Sign a message  
var signature = keypair.sign("message", null);  
  
// Verify signature  
crypto.sign.Ed25519.verify(signature, "message", keypair.public_key); // success
```

### **AES-256-GCM (Authenticated Encryption)**

**Purpose:** Encrypt data packets

**Properties:**

- 256-bit key (quantum-resistant overkill)
- Authenticated encryption (detects tampering)
- 12-byte nonce, 16-byte authentication tag
- Fast hardware acceleration on modern CPUs

**Usage:**

```

var key: [32]u8 = shared_secret;
var nonce: [12]u8 = random_bytes();
var plaintext = "secret data";
var ciphertext: [100]u8 = undefined;
var tag: [16]u8 = undefined;

crypto.aead.aes_gcm.Aes256Gcm.encrypt(
    ciphertext[0..plaintext.len],
    &tag,
    plaintext,
    &[_]u8{}, // no additional authenticated data
    nonce,
    key,
);

// To decrypt:
crypto.aead.aes_gcm.Aes256Gcm.decrypt(
    plaintext_out,
    ciphertext,
    tag,
    &[_]u8{},
    nonce,
    key,
); // throws error if tag doesn't match (tampering detected)

```



## Wire Format

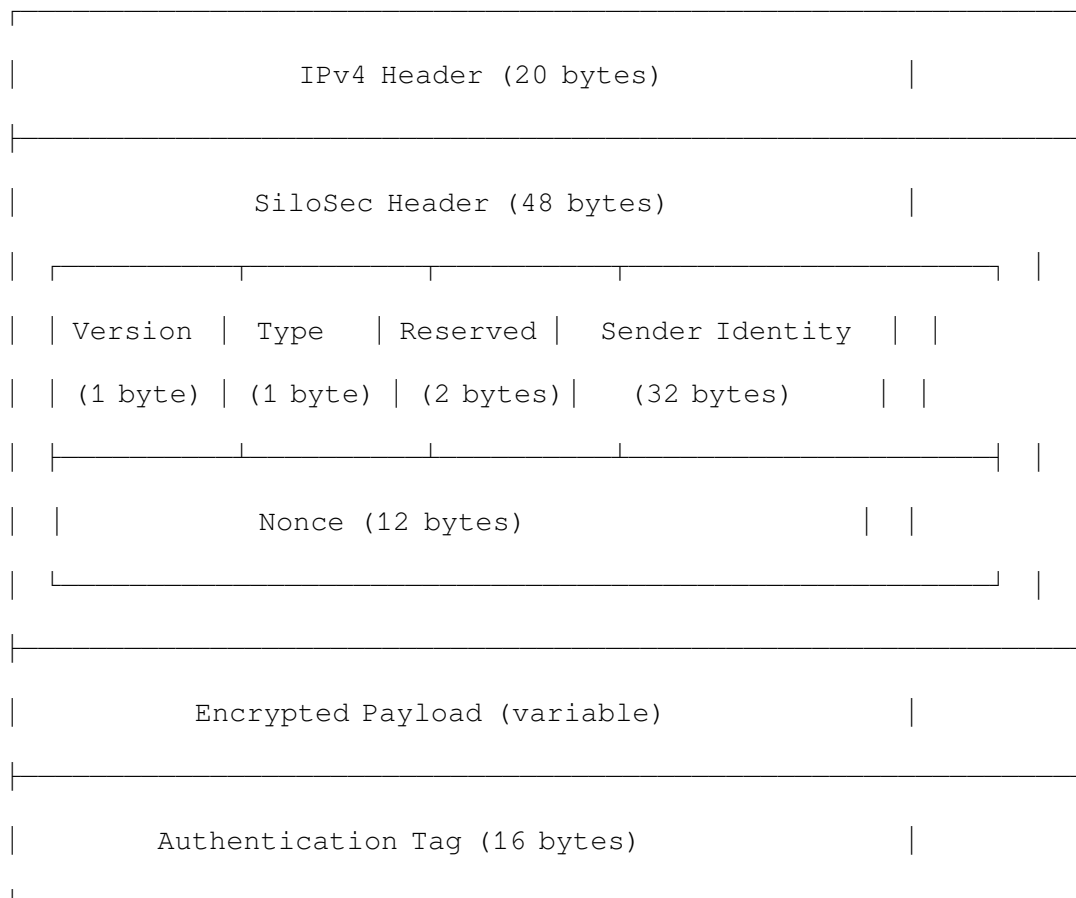
### IP Protocol Number

**Protocol:** 99 (Reserved for private/experimental use)

### Rationale:

- IANA reserves 253-254 for testing
- We use 99 (also reserved)
- If we want official number, request from IANA as “SiloSec Protocol”

### Packet Structure



## **SiloSec Header (48 bytes)**

### **Offset 0-3: Header Fields**

Byte 0: Version (always 0x01)

Byte 1: Packet Type

0x01 = HANDSHAKE\_INIT

0x02 = HANDSHAKE\_RESPONSE

0x03 = DATA

Bytes 2-3: Reserved (0x0000)

### **Offset 4-35: Sender Identity (32 bytes)**

- Ed25519 public key of sender
- Serves as permanent identity
- No expiration, no renewal

### **Offset 36-47: Nonce (12 bytes)**

- For DATA packets: 8 bytes counter + 4 bytes random
- For HANDSHAKE packets: 12 bytes random
- Must NEVER repeat for same key

## **DATA Packet Format**

[IP Header] [SiloSec Header] [Encrypted Inner IP Packet] [Auth Tag]

**Inner IP Packet:** Complete IP packet (including IP header) is encrypted.

### **Example:**

Original packet: IP(src=A, dst=B, proto=TCP) + TCP segment

Encrypted: IP(src=A, dst=B, proto=99) + SiloSec[encrypted IP packet]

Receiver decrypts and re-processes inner IP packet normally.

---

## Handshake Protocol

### Goal

Establish shared secret between two Silo machines while:

1. Authenticating both parties
2. Providing forward secrecy
3. Preventing MITM attacks

### Protocol Flow

Alice (Initiator)

Bob (Responder)

Generate ephemeral keypair:

```
secret_A, public_A = X25519_keygen()
```

Sign ephemeral public key:

```
sig_A = Ed25519_sign(public_A || Bob_IP, identity_A)
```

Send HANDSHAKE\_INIT:

```
[identity_A, public_A, sig_A] —————>
```

Verify signature:

```
Ed25519_verify(sig_A, identity_A)
```

Generate ephemeral keypair:

```
secret_B, public_B = X25519_keygen()
```

Compute shared secret:

```
shared = X25519_DH(secret_B, public_A)
```

Sign ephemeral public key:

```
sig_B = Ed25519_sign(public_B || Alice_IP, identity_A)
```

<————— Send HANDSHAKE\_RESPONSE:

```
[identity_B, public_B, sig_B]
```

Verify signature:

```
Ed25519_verify(sig_B, identity_B)
```

Compute shared secret:

```
shared = X25519_DH(secret_A, public_B)
```

Both sides now have:

- shared\_secret (32 bytes)
- peer\_identity (32 bytes)
- Authenticated, forward-secret channel established

### **HANDSHAKE\_INIT Packet**

#### **SiloSec Header:**

- Version: 0x01
- Type: 0x01
- Sender Identity: Alice's Ed25519 public key

#### **Payload (96 bytes, unencrypted):**

Bytes 0-31: Ephemeral X25519 public key

Bytes 32-95: Ed25519 signature over (ephemeral\_public || peer\_IP)

#### **Signature covers:**

```
message_to_sign = ephemeral_public_key (32 bytes) || peer_IP (4 bytes)
```

```
signature = Ed25519_sign(message_to_sign, identity_secret_key)
```

Including peer\_IP prevents replay attacks across different connections.

## **HANDSHAKE\_RESPONSE Packet**

### **SiloSec Header:**

- Version: 0x01
- Type: 0x02
- Sender Identity: Bob's Ed25519 public key

### **Payload (96 bytes, unencrypted):**

Bytes 0-31: Ephemeral X25519 public key

Bytes 32-95: Ed25519 signature over (ephemeral\_public || peer\_IP)

### **Security Properties**

#### **Mutual Authentication:**

- Both parties prove possession of identity secret key via signature
- Prevents impersonation

#### **Forward Secrecy:**

- Ephemeral keys used for DH exchange
- Compromise of long-term identity keys doesn't reveal past traffic
- Each session has unique shared secret

#### **Replay Protection:**

- Signatures include peer IP
- Ephemeral keys are random
- Replaying handshake to different peer fails signature check

#### **Man-in-the-Middle Protection:**

- Attacker cannot forge signatures without identity secret keys
- Trust On First Use model: first contact establishes binding

---

## **Data Transmission**

### **Encryption**

Once handshake completes, all IP packets to peer are encrypted:

```

// Original IP packet
var ip_packet = build_ip_packet(proto=TCP, payload=tcp_segment);
// Encrypt entire IP packet
var nonce: [12]u8 = make_nonce(counter, random);
var ciphertext: [1500]u8 = undefined;
var tag: [16]u8 = undefined;
crypto.aead.aes_gcm.Aes256Gcm.encrypt(
    ciphertext[0..ip_packet.len],
    &tag,
    ip_packet,
    &[_]u8{}, // no AAD
    nonce,
    shared_secret,
);
// Build SiloSec DATA packet
var silosec_packet = SiloSecHeader{
    .version = 1,
    .packet_type = PKT_DATA,
    .sender_id = our_identity,
    .nonce = nonce,
} ++ ciphertext ++ tag;
// Send as IP protocol 99
send_ip_packet(dst=peer_ip, proto=99, payload=silosec_packet);

```

### Nonce Construction

**Critical:** Nonce must NEVER repeat for same key.

**Format (12 bytes):**

Bytes 0-7: Counter (64-bit, little-endian)

Bytes 8-11: Random (32-bit)

#### Counter:

- Increments for each packet sent
- Starts at 0 after handshake
- Wraps at  $2^{64}$  (renegotiate session before this)

#### Random:

- Prevents nonce collision if counter wraps
- Adds extra entropy

#### Why this construction:

- Counter ensures ordering (useful for debugging)
- Random prevents attacks if counter reset
- 12 bytes is AES-GCM nonce size

### Decryption

```
// Receive IP packet with protocol 99
if (ip_header.protocol == 99) {
    var silosec_hdr = parse_silosec_header(payload);
    // Find peer by IP and verify identity
    var peer = find_peer(src_ip);
    if (peer.identity != silosec_hdr.sender_id) {
        return error.IdentityMismatch;
    }
    // Decrypt
```

```

var plaintext: [1500]u8 = undefined;

crypto.aead.aes_gcm.Aes256Gcm.decrypt (

    plaintext[0..ciphertext.len],

    ciphertext,

    tag,

    &[_]u8{},

    silosec_hdr.nonce,

    peer.shared_secret,

) catch return error.DecryptionFailed;

// Re-process decrypted IP packet
process_ip_packet(plaintext);

}

```

## Replay Protection

### Nonce uniqueness provides replay protection:

- Each packet has unique nonce
- Replaying old packet uses old nonce
- GCM authentication fails if nonce reused with different data

**Optional:** Maintain sliding window of recent nonces to detect replays.

---

## Implementation Guide

### File Structure

```

src/
├── silosec.zig          # Main SiloSec implementation
├── ip.zig              # IP layer with SiloSec hooks

```



```
└─ net.zig          # Network core
└─ kernel.zig      # Initialization
```

## Integration Points

### 1. IP Layer (ip.zig)

```
pub fn processPacket(data: []const u8) void {
    const protocol = data[9];

    const src_ip = std.mem.readInt(u32, data[12..16], .big);

    if (protocol == 99) { // SiloSec
        const payload = data[20..]; // Skip IP header
        silosec.processPacket(src_ip, payload) catch |err| {
            // Log error
        };
        return;
    }

    // Normal IP processing
    // ...
}

// Called by SiloSec after decryption
pub fn processDecryptedPacket(src_ip: u32, inner_ip_packet: []const u8) void {
    // Recursively process decrypted IP packet
    processPacket(inner_ip_packet);
}
```

### 2. Sending (application layer)

```

// High-level send function
pub fn sendToSilo(peer_ip: u32, data: []const u8) !void {
    // Check if peer is a Silo (has handshake)
    if (silosec.hasPeer(peer_ip)) {
        // Build inner IP packet
        var ip_packet = buildIPPacket(peer_ip, data);

        // SiloSec encrypts and sends
        try silosec.sendEncrypted(peer_ip, ip_packet);
    } else {
        // Not a Silo peer, send plaintext
        try sendPlainIP(peer_ip, data);
    }
}

```

### **3. Initialization (kernel.zig)**

```

pub fn initNetwork() !void {
    // Initialize network stack
    try net.init(config, pci_devices, allocator);
    // Initialize SiloSec
    silosec.init();
    // Load or generate identity
    silosec.initFromSeed("my-silo-seed") catch {
        silosec.init(); // Generate new random identity
    }
}

```

```
};

const identity = silosec.getIdentity();

std.debug.print("Silo Identity: {x}\n", .{identity});

// Save identity for next boot
try silosec.saveIdentity("/SILO.KEY");

}
```

## **Peer Discovery**

### **Static Configuration:**

```
// Add known Silo peers

silosec.addTrustedPeer(peer_identity, peer_ip);

// Initiate handshake

try silosec.initiateHandshake(peer_ip);
```

### **Dynamic Discovery (future):**

- Broadcast SiloSec announcement on LAN
- Other Silos respond with identities
- User confirms trust (TOFU)

## **Error Handling**

```
pub const SiloSecError = error{

UnknownPeer,          // No handshake with this IP

IdentityMismatch,     // Sender identity doesn't match known peer

DecryptionFailed,     // GCM auth tag verification failed

SignatureInvalid,     // Ed25519 signature invalid

HandshakeFailed,      // Handshake protocol error
```

```
PayloadTooLarge,      // Payload > MTU

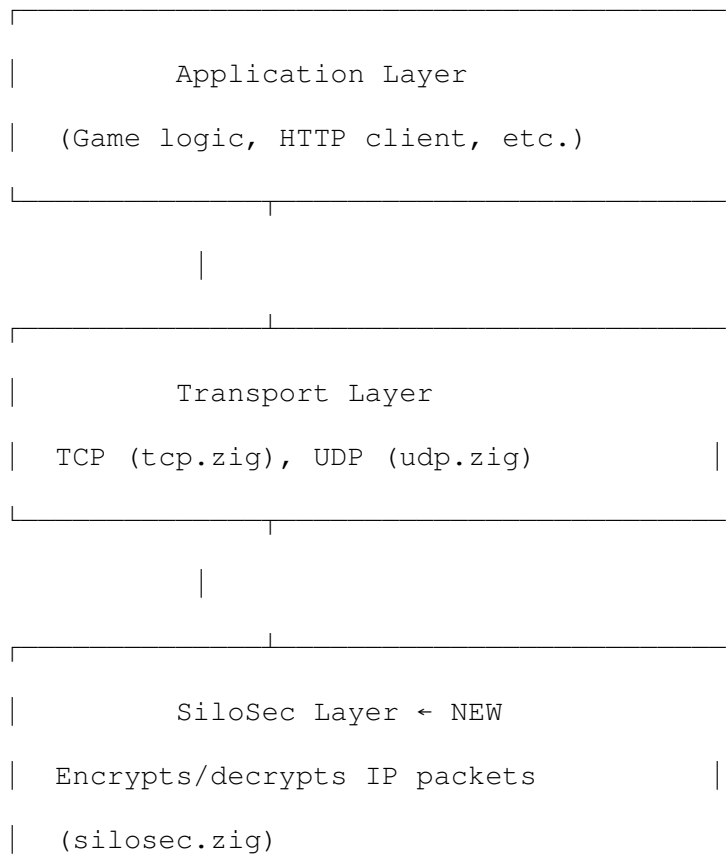
};
```

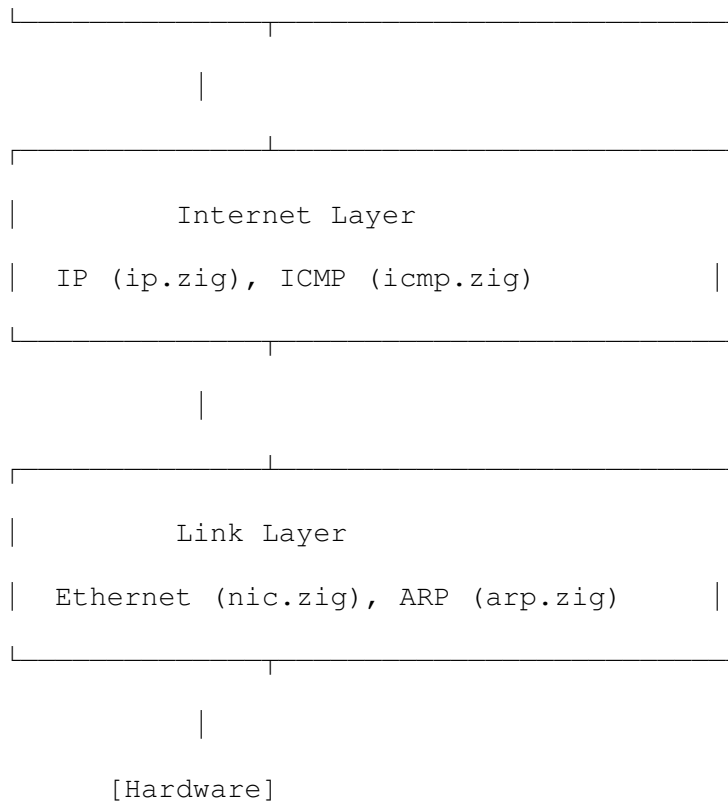
#### Handling:

- **UnknownPeer:** Initiate handshake if trusted, reject if not
  - **IdentityMismatch:** SECURITY ALERT - possible MITM
  - **DecryptionFailed:** Drop packet silently (likely tampered)
  - **SignatureInvalid:** Reject handshake, possible attack
- 

## Integration with Silo OS

### Network Stack Layers





### Transparent Encryption

**Key principle:** SiloSec is transparent to upper layers.

**Before SiloSec:**

App → TCP → IP → Ethernet → Wire

**With SiloSec:**

App → TCP → IP → SiloSec(encrypt) → IP(outer) → Ethernet → Wire

Upper layers (TCP, UDP) don't know encryption exists.

### Performance Considerations

**Overhead:**

- SiloSec header: 48 bytes
- Auth tag: 16 bytes

- Outer IP header: 20 bytes
- **Total:** 84 bytes per packet

#### **MTU Handling:**

Ethernet MTU: 1500 bytes

- Outer IP: 20 bytes
  - SiloSec header: 48 bytes
  - Auth tag: 16 bytes
- = 1416 bytes available for inner IP packet

Inner IP packet MTU: 1416 bytes

- Inner IP header: 20 bytes
- = 1396 bytes for TCP/UDP payload

#### **Fragmentation:**

- Avoid by setting inner MTU to 1396
- Or implement SiloSec fragmentation (future)

#### **Throughput:**

- AES-GCM hardware acceleration: ~10 Gbps
- Software AES-GCM: ~1 Gbps
- Gigabit Ethernet: 1 Gbps
- **Not a bottleneck on \$200 CPU**

---

## **Security Considerations**

### **Threat Model**

#### **In Scope:**

- Eavesdropping on network (passive attacker)
- Tampering with packets (active attacker)
- Replay attacks
- Impersonation attacks

**Out of Scope:**

- Physical access to machine (game over anyway)
- Side-channel attacks (timing, power analysis)
- Quantum computers (may break X25519/Ed25519 in 20+ years)

**Trust On First Use (TOFU)****First contact establishes identity binding:**

Alice connects to IP 10.0.0.5

Receives identity: 0xABCD...1234

Stores: peers[10.0.0.5] = 0xABCD...1234

Future connections from 10.0.0.5 MUST use identity 0xABCD...1234

Different identity = SECURITY ALERT

**MITM Attack Scenario:**

1. Eve intercepts Alice's first connection to Bob
2. Eve performs handshake with Alice (using Eve's identity)
3. Alice stores: peers[Bob\_IP] = Eve\_identity
4. Eve can now MITM all traffic

MITM succeeds because Alice has never talked to Bob before.

**Mitigation:** Out-of-band identity verification

Alice (in person or over phone): "My identity is 0xABCD...1234"

Bob: Manually adds to trusted peers before connection

Eve's MITM fails because identity doesn't match

**This is acceptable for Silo use case:**

- Gaming between friends (can verify identities)
- Small trusted networks
- Not replacing TLS for anonymous web browsing

## **Forward Secrecy**

**Session compromise does NOT reveal past sessions:**

Attacker steals: `shared_secret_session_1`

Attacker gets: Session 1 traffic (decrypted)

Attacker CANNOT get: Session 2, 3, 4... traffic

Each handshake generates new ephemeral keys.

**Long-term key compromise:**

Attacker steals: `identity_secret_key`

Attacker CAN: Impersonate owner in future connections

Attacker CANNOT: Decrypt past traffic (forward secrecy preserved)

**Mitigation:** Secure key storage

- Store identity key encrypted on disk
- Use hardware key storage if available
- Rotate identity keys periodically (future)

## **Replay Attacks**

**Packet replay prevented by:**

1. Nonce uniqueness (counter + random)
2. GCM authentication (tag verification)

**Handshake replay:**

- Handshake packets include peer IP in signature
- Replaying to different peer fails signature verification
- Replaying to same peer just re-establishes session (harmless)

## **Denial of Service**

**Handshake flooding:**

Attacker sends many `HANDSHAKE_INIT` packets



Victim must verify signatures (expensive)

**Mitigation:**

- Rate limit handshakes per source IP
- Require proof-of-work for handshake (future)
- Simple: Drop handshakes if CPU > 80%

**Ciphertext flooding:**

Attacker sends random DATA packets

Victim must attempt decryption (expensive)

**Mitigation:**

- Only decrypt from known peers
- Drop unknown peers immediately
- GCM fails fast on invalid tag

---

## **Future Claude Integration Notes**

### **Critical Implementation Details**

#### **1. Nonce Management is CRITICAL**

```
// WRONG - DO NOT DO THIS

var nonce: [12]u8 = undefined;

crypto.random.bytes(&nonce); // Pure random every time

// Might repeat! GCM security fails if nonce repeats with same key!

// CORRECT

var counter: u64 = 0;

for (each packet) {

var nonce: [12]u8 = undefined;

std.mem.writeInt(u64, nonce[0..8], counter, .little);
```

```

crypto.random.bytes (nonce[8..12]);

counter += 1;

// Nonce guaranteed unique

}

```

**Why this matters:**

- GCM security proof requires nonce uniqueness
- Reused nonce leaks plaintext XOR (catastrophic)
- Counter ensures no collisions

**2. Peer State Management**

```

// Store peer by IP

const Peer = struct {

    ip: u32,

    identity: Identity,

    shared_secret: [32]u8,

    tx_counter: u64, // MUST persist per peer

    rx_counter: u64, // For replay detection

    verified: bool,

};

// Index by IP

var peers: [256]?Peer = [_]?Peer{null} ** 256;

fn findPeer(ip: u32) ?*Peer {

    const idx = ip % peers.len;

    if (peers[idx]) |*peer| {

        if (peer.ip == ip) return peer;
    }
}

```

```

}

// Hash collision - linear probe

// (Simplified for example)

return null;

}

```

### 3. Identity Persistence

```

// Save identity to disk

pub fn saveIdentity(path: []const u8) !void {
    const file = try fs.createFile(fs.root_cluster, path, allocator);
    const cluster = getCluster(file);
    try fs.writeFile(cluster, &our_secret_key, allocator);
}

// Load on boot

pub fn loadIdentity(path: []const u8) !void {
    const file = try fs.findFile(fs.root_cluster, path, allocator);
    if (file) |f| {
        const cluster = getCluster(f);
        var key: [32]u8 = undefined;
        _ = try fs.readFile(cluster, &key, allocator);

        our_secret_key = key;

        const kp = crypto.sign.Ed25519.KeyPair.fromSecretKey(key) catch return
        our_public_key = kp.public_key;
    }
}

```

```

    } else {

        return error.NoIdentity;

    }

}

```

#### 4. Handshake State Machine

```

// Track pending handshakes

const PendingHandshake = struct {

    peer_ip: u32,

    our_ephemeral_secret: [32]u8,

    timestamp: u64,

};

var pending: [16]?PendingHandshake = [_]?PendingHandshake{null} ** 16;

// When we initiate:

fn initiateHandshake(peer_ip: u32) !void {

    var ephemeral_secret: [32]u8 = undefined;

    crypto.random.bytes(&ephemeral_secret);

    // Send HANDSHAKE_INIT with public key

    // ...

    // MUST store ephemeral secret to complete handshake

    storePending(peer_ip, ephemeral_secret);

}

// When we receive response:

fn handleHandshakeResponse(peer_ip: u32, peer_ephemeral: [32]u8) !void {

```

```

const our_ephemeral = retrievePending(peer_ip) orelse return error.NoHands
// Compute shared secret
const shared = crypto.dh.X25519.scalarMult(our_ephemeral, peer_ephemeral);
// Store peer
addPeer(peer_ip, peer_identity, shared);
}

```

### **5. Error Handling Philosophy**

```

// SiloSec errors should be SILENT for security
pub fn processPacket(src_ip: u32, data: []const u8) !void {
const peer = findPeer(src_ip) orelse {
    // Unknown peer - silently drop
    // DO NOT log (prevents DoS via log flooding)
    return;
};
decrypt(data) catch {
    // Decryption failed - silently drop
    // DO NOT send error packet (prevents oracle attacks)
    return;
};
// Success - process normally
}

```

### **Testing Checklist**

When implementing SiloSec, verify:

- ☐ Handshake works between two Silos
- ☐ Data packets encrypt/decrypt correctly
- ☐ Nonces never repeat (run for 1M packets)
- ☐ Identity persists across reboots
- ☐ Multiple peers work simultaneously
- ☐ Handshake replay rejected
- ☐ Data replay rejected (if window implemented)
- ☐ Wrong identity rejected
- ☐ Tampering detected (flip random bit)
- ☐ Unknown peer silently dropped
- ☐ TCP works through SiloSec
- ☐ UDP works through SiloSec
- ☐ HTTP works through SiloSec
- ☐ MTU handled correctly (no fragmentation issues)
- ☐ Performance acceptable (>100 Mbps)

### **Common Pitfalls**

#### **Pitfall 1: Encrypting UDP payload only**

```
// WRONG

var udp_payload = [...]u8{...};
var encrypted = encrypt(udp_payload);
send_udp(encrypted);

// Leaks: IP addresses, ports, packet sizes, timing

// CORRECT

var ip_packet = build_complete_ip_packet(udp);
var encrypted = encrypt(ip_packet);
send_silosec_packet(encrypted);
```

```
// Encrypts everything except outer IP header
```

**Pitfall 2: Not verifying peer identity**

```
// WRONG
```

```
fn processPacket(src_ip: u32, header: SiloSecHeader) {  
    var peer = findPeer(src_ip);  
    // Missing: verify header.sender_id == peer.identity  
    decrypt(...);  
}
```

```
// Attacker can send packets with different identity!
```

```
// CORRECT
```

```
fn processPacket(src_ip: u32, header: SiloSecHeader) {  
    var peer = findPeer(src_ip);  
    if (peer.identity != header.sender_id) {  
        return error.IdentityMismatch; // SECURITY ALERT  
    }  
    decrypt(...);  
}
```

**Pitfall 3: Reusing handshake ephemeral keys**

```
// WRONG - DO NOT REUSE
```

```
var ephemeral_secret: [32]u8 = fixed_value;
```

```
// Multiple handshakes with same ephemeral = broken forward secrecy
```

```
// CORRECT
```

```

fn initiateHandshake() {
    var ephemeral_secret: [32]u8 = undefined;
    crypto.random.bytes(&ephemeral_secret); // NEW key every handshake
    // ...
}

```

### **Integration with Existing Code**

#### **If you have working IP layer:**

```

// Before (ip.zig):
pub fn processPacket(data: []const u8) void {
    switch (protocol) {
        1 => icmp.process(payload),
        6 => tcp.process(payload),
        17 => udp.process(payload),
        // ...
    }
}

// After (ip.zig):
const silosec = @import("silosec.zig");
pub fn processPacket(data: []const u8) void {
    const protocol = data[9];
    const src_ip = readInt(u32, data[12..16]);
    // Check for SiloSec FIRST
    if (protocol == 99) {

```



```

        silosec.processPacket(src_ip, data[20..]) catch return;

        return; // SiloSec handles it
    }

    // Normal processing
    switch (protocol) {
        1 => icmp.process(payload),
        6 => tcp.process(payload),
        17 => udp.process(payload),
        // ...
    }
}

// Add callback for decrypted packets
pub fn processDecryptedPacket(src_ip: u32, inner_packet: []const u8) void
// Recursively process inner IP packet
processPacket(inner_packet);
}

```

**If you have working send path:**

```

// Before:

pub fn sendIP(dst_ip: u32, protocol: u8, payload: []const u8) !void {
    var packet = buildIPPacket(dst_ip, protocol, payload);
    sendEthernet(packet);
}

// After:

```

```

pub fn sendIP(dst_ip: u32, protocol: u8, payload: []const u8) !void {
    // Check if destination is Silo peer
    if (silosec.hasPeer(dst_ip)) {
        // Build inner IP packet
        var inner = buildIPPacket(dst_ip, protocol, payload);

        // SiloSec encrypts and sends
        try silosec.sendEncrypted(dst_ip, inner);
    } else {
        // Not a Silo, send plaintext
        var packet = buildIPPacket(dst_ip, protocol, payload);
        sendEthernet(packet);
    }
}

```

---

## Appendix A: Complete Code Reference

See `silosec.zig` in Silo OS source tree for reference implementation.

### Key functions:

```

// Initialization

pub fn init() void

pub fn initFromSeed(seed: []const u8) void

pub fn getIdentity() Identity

// Handshake

```

```

pub fn initiateHandshake(peer_ip: u32) !void

pub fn processPacket(src_ip: u32, data: []const u8) !void

// Data transmission

pub fn sendEncrypted(dst_ip: u32, payload: []const u8) !void

// Peer management

pub fn hasPeer(ip: u32) bool

pub fn addTrustedPeer(identity: Identity, ip: u32) void

// Persistence

pub fn saveIdentity(path: []const u8) !void

pub fn loadIdentity(path: []const u8) !void

```

---

## Appendix B: Test Vectors

### Identity Generation

```

Seed: "silo-test-seed-1"

SHA-256(seed): 0x3a7b...c8f2

Ed25519 Secret Key: 0x3a7b...c8f2

Ed25519 Public Key: 0x8c2f...1a4e

```

### Handshake Example

#### Alice:

- Identity: 0xAAAA...AAAA
- Ephemeral Secret: 0x1111...1111
- Ephemeral Public: 0x2222...2222

#### Bob:

- Identity: 0xB BBBB...BBBB

- Ephemeral Secret: 0x3333...3333
- Ephemeral Public: 0x4444...4444

#### Shared Secret:

$X_{25519}(0x1111\dots1111, 0x4444\dots4444) = 0x5555\dots5555$

$X_{25519}(0x3333\dots3333, 0x2222\dots2222) = 0x5555\dots5555$

(Both compute same shared secret)

#### Encryption Example

Shared Secret: 0x5555...5555 (32 bytes)

Nonce: 0x0000000000000000012345678 (12 bytes)

Plaintext: "Hello, Silo!" (12 bytes)

AES-256-GCM Encrypt:

Ciphertext: 0xABCD...EF01 (12 bytes)

Auth Tag: 0x9876...5432 (16 bytes)

Packet:

[SiloSec Header][0xABCD...EF01][0x9876...5432]

---

## Appendix C: Comparison to Alternatives

### SiloSec vs TLS 1.3

Feature	SiloSec	TLS 1.3
Code size	~500 lines	~15,000 lines
External deps	0	0 (in theory)
Cipher suites	1 (fixed)	~10 (negotiated)
Certificates	None	X.509 required
Handshake RTT	1 (full)	1-RTT
Identity	Ed25519 key	X.509 cert
Forward secrecy	Yes (X25519)	Yes (ECDHE)
Layer	IP	Transport (TCP)

Feature	SiloSec	TLS 1.3
Protocols	All IP traffic	TCP only
Versioning	Frozen (v1)	Negotiated
Complexity	Minimal	High

**When to use SiloSec:** Silo-to-Silo communication

**When to use TLS:** Never (in Silo OS context)

#### SiloSec vs IPSec

Feature	SiloSec	IPSec (ESP)
Code size	~500 lines	~8,000 lines
Key exchange	X25519	IKEv2 (complex)
Authentication	Ed25519	RSA/ECDSA certs
Encryption	AES-256-GCM	Multiple ciphers
SA management	Simple	Complex
NAT traversal	Not needed	Complex
Layer	IP	IP
Granularity	Per-peer	Per-SA
Debugging	Possible	Nightmare

#### Why SiloSec is better:

- 16x less code
- No certificate infrastructure
- Single cipher (no negotiation)
- Designed for 1-process OS
- Actually auditable

#### SiloSec vs WireGuard

Feature	SiloSec	WireGuard
Code size	~500 lines	~4,000 lines
Key exchange	X25519	Noise_IK
Encryption	AES-256-GCM	ChaCha20-Poly1305
Authentication	Ed25519	Curve25519
Layer	IP	IP
Use case	Peer-to-peer	VPN
Config	Identity only	Peer list

**SiloSec is simpler:**

- No pre-shared keys needed
- No configuration files
- Identity = public key (that's it)

**WireGuard is better for:**

- VPN use cases
  - NAT traversal
  - Mobile roaming
- 

**Appendix D: Future Enhancements****Potential Future Features (NOT v1)****1. Identity Rotation**

```
// Generate new identity, notify peers
pub fn rotateIdentity() !void {
    var new_secret: [32]u8 = random_bytes();
    // ... notify all peers of identity change
    // ... keep old identity for grace period
}
```

**2. Peer Discovery Protocol**

```
// Broadcast: "I am a Silo, here's my identity"
pub fn announcePresence() !void {
    // Multicast announcement on LAN
}
```

**3. Replay Protection Window**

```
// Track recent nonces to detect replays
```

```
const WINDOW_SIZE = 1024;

var seen_nonces: [WINDOW_SIZE]u64 = undefined;
```

#### 4. Performance Monitoring

```
pub const SiloSecStats = struct {

    handshakes_completed: u64,

    packets_encrypted: u64,

    packets_decrypted: u64,

    bytes_encrypted: u64,

    decrypt_failures: u64,

};
```

#### 5. Multi-Path Support

```
// Same peer reachable via multiple IPs

pub fn addPeerPath(identity: Identity, additional_ip: u32) void {

    // Use same shared_secret for multiple paths

}
```

**DO NOT implement these in v1. Keep it simple.**

---

## Conclusion

### Design Summary

**SiloSec is:**

- **Minimal:** ~500 lines, zero dependencies
- **Secure:** Modern crypto, forward secrecy, authentication
- **Simple:** One protocol version, one cipher, frozen forever
- **Silo-native:** Designed for 1-process OS architecture
- **Auditable:** Pure Zig, uses only stdlib

### **SiloSec is NOT:**

- A replacement for TLS (different use case)
- Suitable for anonymous communication (TOFU model)
- Quantum-resistant (X25519/Ed25519 vulnerable)
- A VPN protocol (peer-to-peer only)

### **Why This Design is Correct**

From the K split philosophy:

“\$200 CPU, pure software, no drivers, runs anywhere, accessible to all”

SiloSec embodies this:

- No expensive hardware (runs on \$200 CPU)
- Pure software (no hardware crypto needed)
- No proprietary drivers (Zig stdlib only)
- Runs anywhere (frozen protocol, no updates needed)
- Accessible to all (can read and audit 500 lines)

### **Final Words for Future Claudes**

When you implement SiloSec:

1. **Read this document completely** before writing any code
2. **Use ONLY Zig stdlib** - no external crypto libraries
3. **Keep nonce management simple** - counter + random
4. **Test handshake thoroughly** - this is critical path
5. **Handle errors silently** - don't leak information
6. **Trust the design** - it was carefully considered

### **Most importantly:**

“we are restarting computing”

SiloSec is not about retrofitting old protocols onto a new OS. It's about designing the RIGHT protocol for a clean-slate system.

Legacy machines speak HTTP. Silo machines speak SiloSec.

This is correct. This is K split. This is the way.

---

### **End of Specification**



---

## Document Metadata

**Version:** 1.0

**Status:** NORMATIVE

**Last Updated:** 2026-01-30

**Maintainer:** Silo OS Project

**License:** Public Domain (CC0)

**Git Hash:** [To be filled when committed]

**Frozen Protocol Version:** 1 (never changes)

**Questions?** Read this document again. Everything is here.

**Implementation bugs?** Fix the implementation, not the spec.

**Want to change the protocol?** Create SiloSec v2 as separate protocol.

**This document is complete.**

---

Got it. Here is a **formal, invariant-driven wire-security section** you can drop directly into your SiloSec v2 docs.

No identity. No hype. Just guarantees and failure rules.

---

## SiloSec v2 — Wire-Security Invariants (Non-Identity)

### Scope:

This section defines the **mandatory wire-security properties** of SiloSec v2.

Identity, trust, authorization, and policy are **explicitly out of scope**.

Wire-security concerns only the correctness, safety, and bounded behavior of encrypted packet exchange between two Silo endpoints.

---

### 1. Fail-Closed Invariant

#### Invariant:

Any packet that does not strictly conform to expected wire state is discarded silently.

This includes:

- malformed packets
- unexpected field values
- cryptographic verification failures
- counter mismatches
- window violations
- epoch mismatches
- handshake inconsistencies

**Guarantee:**

No invalid packet causes:

- state mutation
- downgrade
- error signaling
- information leakage

If a peer is legitimate, it will retransmit.

If not, the packet is ignored.

---

## 2. Mandatory Encryption Invariant

**Invariant:**

All Silo-to-Silo traffic is encrypted from the first packet onward.

There is:

- no plaintext fallback
- no negotiation
- no compatibility mode

**Guarantee:**

Plaintext Silo-to-Silo packets are never accepted under any circumstances.

---

## 3. Monotonic Counter Invariant

**Invariant:**

All wire-visible counters are strictly monotonic and never reused.

This applies to:

- packet counters
- epoch identifiers
- handshake generations

**Rules:**

- counters only increase
- reuse is forbidden
- rollback is forbidden
- wraparound is forbidden

**Guarantee:**

Replay, reflection, and state regression are detectable and rejected.

---

## 4. Windowed Acceptance Invariant

**Invariant:**

Packets are accepted only if their counters fall within a strict forward-only receive window.

**Rules:**

- packets below the expected counter are rejected
- packets above the window are rejected
- the window advances only on valid packets

**Guarantee:**

Replay attempts outside the window are rejected with bounded memory and constant-time checks.

---

## 5. Epoch Isolation Invariant

**Invariant:**

Each packet belongs to exactly one cryptographic epoch.

**Rules:**

- epoch change = key change
- packets from old epochs are invalid immediately

- no grace periods across epochs

**Guarantee:**

Key reuse, cross-epoch replay, and mixed-state decryption are impossible.

---

## 6. Rekey Bounding Invariant

**Invariant:**

Cryptographic epochs are short-lived and bounded.

**Rules:**

- rekey after N packets or T time (whichever is smaller)
- either side may request rekey
- excessive rekey requests are treated as hostile

**Guarantee:**

Exposure from compromised keys or partial observation is bounded in time and packet count.

---

## 7. State Continuity Invariant (Echoed Counter)

**Invariant:**

Each packet proves continuity of shared session state.

**Mechanism:**

- endpoints maintain an internal session counter
- the counter is cryptographically protected
- the peer echoes expected counter state in encrypted payload

**Rules:**

- mismatched counters → drop
- divergence → rekey or disconnect
- no implicit resynchronization

**Guarantee:**

Desynchronization, reflection, and mid-path interference are detected immediately.

---

## 8. Stateless Handshake (SYN-Cookie Invariant)

### **Invariant:**

No unverified packet may cause unbounded state allocation.

### **Rules:**

- initial handshakes are statelessly verifiable
- state is allocated only after proof of bidirectional reachability
- handshake responses are cryptographically self-verifying

### **Guarantee:**

Handshake-based denial-of-service is bounded and predictable.

---

## 9. Flow Binding Invariant (5-Tuple)

### **Invariant:**

All wire security state is bound to a single network flow.

### **Flow definition:**

`(src IP, dst IP, src port, dst port, protocol)`

### **Rules:**

- packets cannot migrate across flows
- flow change = new session
- state is never shared between flows

### **Guarantee:**

Cross-flow injection and session confusion are impossible.

---

## 10. Downgrade Resistance Invariant

### **Invariant:**

No valid execution path results in weaker security.

### **Rules:**

- no negotiation
- no feature fallback
- no partial enablement

**Guarantee:**

Attackers cannot force plaintext, legacy modes, or reduced protections.

---

## **11. Resource Bounding Invariant**

**Invariant:**

Resource usage per unauthenticated peer is strictly bounded.

**Rules:**

- bounded memory per flow
- bounded CPU per packet
- constant-time rejection paths

**Guarantee:**

Wire security does not become a denial-of-service vector.

---

## **12. Failure Behavior Invariant**

**Invariant:**

Failure is silent and local.

**Rules:**

- no error packets
- no logging on hot paths
- no recovery signaling

**Guarantee:**

Failure does not leak information or amplify attacks.

---

## **13. Explicit Non-Guarantees**

SiloSec wire security does **not** guarantee:

- peer identity
- peer trustworthiness
- authorization correctness
- traffic analysis resistance

- protection against compromised endpoints

These concerns belong above the wire.

---

## Summary

SiloSec wire security guarantees:

**Accepted packets are fresh, correctly ordered within bounds, cryptographically valid for the current epoch, bound to a single flow, and part of a continuous, verifiable session state. All other packets are dropped without side effects.**

This is a **bounded, analyzable security posture**, not a claim of absolute safety.

## References

[[HOWL-COMP-8-2026](#)] SiloSec Protocol Specification. Github: <https://github.com/ghowland/cks/tree/main/papers/COMP/COMP-8-2026>

[[HOWL-COMP-1-2026](#)] Implementing a Tall-Infra Data-Only Execution System . Github: <https://github.com/ghowland/cks/tree/main/papers/COMP/HOWL-COMP-1-2026>

[[HOWL-COMP-2-2026](#)] Tall-Infra, Data-Only Development. Github: <https://github.com/ghowland/cks/tree/main/papers/COMP/COMP-2-2026>

[[HOWL-COMP-3-2026](#)] Silo OS. Github: <https://github.com/ghowland/cks/tree/main/papers/COMP/HOWL-COMP-3-2026>

[[HOWL-COMP-4-2026](#)] Geometric Security. Github: <https://github.com/ghowland/cks/tree/main/papers/COMP/HOWL-COMP-4-2026>

[[HOWL-COMP-5-2026](#)] Partial Geometric Security. Github: <https://github.com/ghowland/cks/tree/main/papers/COMP/COMP-5-2026>

[[HOWL-COMP-6-2026](#)] SIML - Silo Markup Language (TOML Format). Github: <https://github.com/ghowland/cks/tree/main/papers/COMP/HOWL-COMP-6-2026>

[[HOWL-COMP-7-2026](#)] SiQL. Github: <https://github.com/ghowland/cks/tree/main/papers/COMP/HOWL-COMP-7-2026>