Authenticated Encrypted Relay Network - AERN

Revision 1.1c, May 22, 2026

John G. Underhill – john.underhill@protonmail.com

This document is an engineering level description of the AERN authenticated and encrypted relay network. This document describes the network protocol AERN, an anonymous encrypted proxy network that enables secure anonymous communications between client devices.

Contents

Foreword....5
1. Introduction....6
1.1 Problem Description:....7
1.2 Design Requirements:....7
1.3 Purpose....8
2. Scope....9
2.1 Application....9
3. References....10
3.1 Normative References....10
4. Terms and Definitions....11
4.1 Cryptographic Primitives....11
4.2 Network References....11
5. Protocol Description....14
5.1 Objectives....14
5.2 Key Components and Their Roles....15
5.3 Network Initialization....16
5.4 Network Initialization....17
5.4.1 Root certificate creation....18
5.4.2 ADC Initialization....20
5.4.4 APS Initialization....23
5.4.5 Client Initialization....28
5.4.6 Client Entry Node Selection and Connection....30
5.4.7 Route Map Creation....32
5.4.8 Network Traversal and Relay Packet Format....33
5.4.8.1 Encrypted Relay Payload Header....34
5.4.8.2 Ingress-to-Egress Relay Session Establishment....35
5.4.8.3 Backend Transport Interface....36
5.4.8.4 Relay Fragmentation and Reassembly....37

5.4.8.5 Dummy Relay Traffic and Ingress Delay 37
5.4.9 Cipher Re-instantiation, Replay Protection, and Forward Security 38
5.4.10 Disconnects, Errors, and Session Teardown 38
6. Mathematical Description 40
6.1 Converge Broadcast 40
6.2 Incremental Update 42
6.3 Master Encryption Key Exchange 44
6.4 Register Update Request 47
6.5 Remote Signing Request 48
6.6 Resign Request 49
6.7 Revoke Broadcast 50
6.8 Join Request 51
6.9 Relay Packet Encryption 53
6.10 Route Map Forwarding 53
6.11 Relay Session-Open State Machine 54
6.12 Relay Fragmentation 54
6.13 Dummy Packet Generation 54
6.14 Replay and Freshness 55
6.15 Backend Transport Boundary 55
6.16 Announce Broadcast and Response 55
6.17 Converge Update 56
6.18 Fragment-Key Request and Response 56
6.19 Register Request and Response 57
6.20 Register Update Versioned Response 58
6.21 Topological Query Request and Response 59
6.22 Topological Status Request and Response 59
7. Security Analysis 61
7.1 Introduction to Security Context 61
7.2 Comparative Analysis with TOR 61

7.3 Security Strengths in Private Implementation....61
7.4 Node Authentication and Topological Integrity....61
7.5 Channel Confidentiality and Integrity....62
7.6 Replay Resistance....62
7.7 Relay Session Security....62
7.8 Fragmentation Security....62
7.9 Dummy Traffic and Timing Mitigation....62
7.10 Backend Transport Boundary....62
7.11 Potential Limitations and Mitigations....63
7.12 Conclusion of Security Analysis....63
8. Cryptanalysis of the Authenticated Encrypted Relay Network....64
8.1 Adversary Model and Security Targets....64
8.2 Cryptographic Core....64
8.3 Handshake-Level Analysis....64
8.4 Data-Phase Cryptanalysis....64
8.5 Replay and Freshness Analysis....65
8.6 Anonymity and Traffic-Analysis Resistance....65
8.7 Dummy Traffic Limits....65
8.8 Comparison with Established Anonymous Overlays....65
8.9 Potential Weaknesses and Mitigations....66
8.10 Conclusion of Cryptanalysis....66
9. Conclusion....67
Appendix A. AERN Conformance Profile....70

Foreword

This document is intended as the preliminary draft of a new standards proposal, and as a basis from which that standard can be implemented. We intend that this serves as an explanation of this new technology, and as a complete description of the protocol. This document is the first revision of the specification of AERN, further revisions may become necessary during the pursuit of a standard model, and revision numbers shall be incremented with changes to the specification. The reader is asked to consider only the most recent revision of this draft, as the authoritative expression of the AERN specification. The inventor and author of this specification is John G. Underhill, and can be reached at contact@qrcscorp.ca

1. Introduction

AERN (Authenticated Encrypted Relay Network) is a proxy chaining protocol, one that uses a fully meshed ‘cloud’ of proxy servers, that provides message authentication and encryption, and a network system that also provides a strong guarantee of anonymity to the users.
Each proxy node on the network undergoes an asymmetric key exchange with every other proxy, exchanging a shared secret that is expanded and used to key two symmetric cipher instances; one for the send channel the other for the receive channel, creating a bi-directional encrypted tunnel. These tunnels are used to encrypt and decrypt packet flows as they travel between proxy servers. These proxy server routes are randomly assembled circuits; proxy servers selected at random, and a random ranged number of nodes in the path.
The network is administrated over by a domain controller which handles device registrations and control messages, and a root server which acts as the authentication trust anchor.
An entry node is chosen at random by the client from a list of proxy servers it receives during network registration. The device performs an asymmetric key exchange with the entry node and establishes an encrypted tunnel to access the proxy network.
Once a network of proxy nodes is ‘synchronized’, and they all share encrypted tunnels with each other, all data traversing these nodes is encrypted using symmetric encryption, which is computationally cheap, and allows for a long proxy chain without incurring delay or jitter associated with video or voice messaging.
Egress nodes perform the terminal relay function for outbound traffic. In AERN, the egress APS authenticates and decrypts the relay packet, validates the relay session, reassembles fragments when required, and delivers the recovered serialized packet to a configured backend transport callback. The backend may bind AERN to a TUN or TAP device, raw socket path, userspace TCP or UDP stack, or application transport adapter.
Every packet sent through the proxy circuit is the same size; 1500 bytes, the standard network MTU, and so packets remain size-indistinguishable as they traverse the proxy circuit.
Packets are encrypted using different encrypted tunnel interfaces between proxies, the symmetric cipher (RCS) is a CTR based authenticated stream cipher, so the same packet travelling through the same circuit at different times will still be a unique and indistinguishable ciphertext. The entire message packet is encrypted by the encrypted tunnels between proxy nodes, so that nothing is identifiable or trackable in that message. Given the number of nodes is random; between three (*entry node – forwarding node – exit node*) and a tunable maximum hops value (default of sixteen), timing variances are difficult to establish, and on a network under load, almost impossible to calculate.
Packets that exceed the MTU size are fragmented and reassembled at the destination. The path changes every time a packet is sent from source to destination, whether from client to server or server to client; a new random path, that maintains the source and destination nodes in the path but changes intermediate nodes is calculated, randomizing the route for every packet that traverses the network.
There can be any number of proxy nodes in the network up to a theoretical maximum, but in practical terms, a 100 or more nodes is reasonable, with more nodes representing more path possibilities and increased resistance to path calculation or correlating clients with exit traffic.
AERN operates more like a modern VPN service than a TOR network, in that AERN is not a single global-scale network, but domain based, where multiple AERN domains can be added to a federation of proxy networks. Different AERN domains can be listed in the VPN applications choice window, selected at random from a list of domain options, and even switched and rotated during a client session. AERN is a truly anonymous form of VPN, one which not only hides address information from the destination server, but strongly mitigates traffic flow observations by an outside observer,

providing anonymity in the broader sense, against attempts to profile individuals and monitor their internet traffic.
AERN defines inter-domain gateway reservation structures, including aern_idg_hint and aern_idg_certificate, and an IDG application port assignment. These structures are reserved for future inter-domain gateway operation. They SHALL NOT be treated as part of the active single-domain relay path unless a deployment enables and specifies the IDG service profile.

1.1 Problem Description:

The US Naval Research Laboratory (NRL) began developing TOR (The Onion Router) in 1995, as a means to protect US intelligence traffic transiting the public internet. TOR uses layered encryption to wrap a message stream as it traverses across intermediate proxy nodes, decrypting a layer each time it traverses through a node in the path. In this way the message changes between nodes, and the route the message takes along the path (three nodes in modern TOR implementations) is obscured, providing anonymity by concealing which devices are communicating with each other. This concealment of metadata allows for devices to communicate without the source and destination being correlated, and can also allow for secure communications to destinations that are being blocked or actively monitored by signals intelligence agencies. TOR has been used by people in nations where the government blocks access to foreign media, by journalists and civil rights advocates to communicate in places where a government may be hostile to human rights, in authoritarian regimes where human rights are abused and strongly restricted. It has become an important tool to grass roots democratic movements, and provides a safe and anonymous communications platform to many people important in the advocacy of freedom and human rights around the world. TOR has also been targeted by signals intelligence agencies, both in nations struggling under authoritarianism and in the Western world, where attacks on the efficacy of these secure networks is ongoing and represents a serious compromise to the continued reliability and sustainability of the TOR network. Many serious attacks against TOR have evolved over the past twenty years; exit node monitoring, traffic correlation attacks, malicious nodes, guard node identification, Sybil attacks, directory authority compromise, and soon threats from quantum computers to the core cryptographic algorithms. Simply stated, a more futuristic and sustainable alternative to TOR, a new network system must be developed, one which counters the threats that have evolved since its inception, and can be used to create an encrypted, authenticated, and truly anonymous network system that can withstand future threats.

1.2 Design Requirements:

The distributed security system is computationally economical, with functions in the primary key exchange and tunnel being performed solely by symmetric cryptography.
That asymmetric functions be constrained to network control messaging, and device registration and initialization.
Certificates are used as a means to authenticate devices and the messages they produce during device initialization and network operations. Each device generates its own asymmetric signature key-pair, and retains the secret signing key. Each device uses the signature verification key to create a certificate which must be signed by the root security server, the trust anchor for the proxy network.

The network must be scalable, expensive asymmetric operations must be constrained to registration and key exchange with participating devices, after which operations become administrative, and devices use the minimal network and hardware resources to function.
The system must be designed to be a form of encrypted tunneling with no tolerance for failure. Any failure in the exchange between nodes in the scheme, whether it be authentication or the distribution of keys, packet values, or symmetric or asymmetric authentication failure, causes the failure of the exchange, and the collapse of the circuit.

1.3 Purpose

AERN provides an authenticated and encrypted tunnel routed across multiple autonomous devices.

The AERN cryptosystem, has been designed in such a way that:

1) The system provides a strong degree of anonymity against metadata harvesting techniques, using an end-to-end encrypted network tunneling protocol.
2) Uses an advanced authentication system, across multiple core devices, and a hierarchal certificate scheme for authentication.
3) That the model must be scalable, computationally efficient, and provide strong security guarantees against a wide range of classical and quantum attacks.
4) That the system must be functionally anonymous in all aspects, that it keep no logs or metadata about traffic flow.

2. Scope

This document describes the AERN (Authenticated Encrypted Relay Network) protocol, which is used to establish an authenticated, encrypted, and anonymous duplexed communications stream between two devices. The protocol is described in this document, and references to the example C implementation are available, including specific settings and software components necessary to its design.
The AERN protocol has been designed as an anonymous network composed of autonomous proxy servers acting in a fully meshed cloud network, that provides both end-to-end authenticated encryption and hides traffic flow destinations from observation.

2.1 Application

The AERN protocol is intended for institutions that implement secure communication streams used to encrypt and authenticate secret information exchanged between client devices.
The network design, key exchange functions, authentication and encryption of messages, and control message exchanges between devices defined in this document must be considered as mandatory elements in the construction of an AERN network. Components that are not necessarily mandatory, but are the recommended settings or usage of the protocol will be denoted by the key-words **SHOULD**. In circumstances where strict conformance to implementation procedures is required but not necessarily obvious, the key-word **SHALL** will be used to indicate compulsory compliance is required to conform to the specification, likewise warnings indicating changes to the specification that are prohibited will be notated with **SHALL NOT**.

3. References

3.1 Normative References

3.1.1 FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable Output Functions: This standard specifies the SHA-3 family of hash functions, including SHAKE extendable-output functions. https://doi.org/10.6028/NIST.FIPS.202

3.1.2 FIPS 203: Module-Lattice-Based Key Encapsulation Mechanism (ML-KEM): This standard specifies ML-KEM, a key encapsulation mechanism designed to be secure against quantum computer attacks. https://doi.org/10.6028/NIST.FIPS.203

3.1.3 FIPS 204: Module-Lattice-Based Digital Signature Standard (ML-DSA): This standard specifies ML-DSA, a set of algorithms for generating and verifying digital signatures, believed to be secure even against adversaries with quantum computing capabilities. https://doi.org/10.6028/NIST.FIPS.204

3.1.4 NIST SP 800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash, and ParallelHash: This publication specifies four SHA-3-derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash. https://doi.org/10.6028/NIST.SP.800-185

3.1.5 NIST SP 800-90A Rev. 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators: This publication provides recommendations for the generation of random numbers using deterministic random bit generators. https://doi.org/10.6028/NIST.SP.800-90Ar1

3.1.6 NIST SP 800-108: Recommendation for Key Derivation Using Pseudorandom Functions: This publication offers recommendations for key derivation using pseudorandom functions. https://doi.org/10.6028/NIST.SP.800-108

3.1.7 FIPS 197: The Advanced Encryption Standard (AES): This standard specifies the Advanced Encryption Standard (AES), a symmetric block cipher used widely across the globe.
https://doi.org/10.6028/NIST.FIPS.197

4. Terms and Definitions

4.1 Cryptographic Primitives

4.1.1 Kyber

The Kyber asymmetric cipher and NIST Post Quantum Competition winner.

4.1.2 McEliece

The McEliece asymmetric cipher and NIST Round 3 Post Quantum Competition candidate.

4.1.3 Dilithium

The Dilithium asymmetric signature scheme and NIST Post Quantum Competition winner.

4.1.5 SPHINCS+

The SPHINCS+ asymmetric signature scheme and NIST Post Quantum Competition winner.

4.1.6 RCS

The wide-block Rijndael hybrid authenticated symmetric stream cipher.

4.1.7 SHA-3

The SHA-3 hash function NIST standard, as defined in the NIST standards document FIPS-202; SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.

4.1.8 SHAKE

The NIST standard Extended Output Function (XOF) defined in the SHA-3 standard publication FIPS-202; SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.

4.1.9 KMAC

The SHA-3 derived Message Authentication Code generator (MAC) function defined in NIST special publication SP800-185: SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash.

4.2 Network References

4.2.1 Bandwidth

The maximum rate of data transfer across a given path, measured in bits per second (bps).

4.2.2 Byte

Eight bits of data, represented as an unsigned integer ranged 0-255.

4.2.3 Certificate

A digital certificate, a structure that contains a signature verification key, expiration time, and serial number and other identifying information. A certificate is used to verify the authenticity of a message signed with an asymmetric signature scheme.

4.2.4 Domain

A virtual grouping of devices under the same authoritative control that shares resources between members. Domains are not constrained to an IP subnet or physical location but are a virtual group of devices, with server resources typically under the control of a network administrator, and clients accessing those resources from different networks or locations.

4.2.5 Duplex

The ability of a communication system to transmit and receive data; half-duplex allows one direction at a time, while full-duplex allows simultaneous two-way communication.

4.2.6 Gateway: A network point that acts as an entrance to another network, often connecting a local network to the internet.

4.2.7 IP Address

A unique numerical label assigned to each device connected to a network that uses the Internet Protocol for communication.

4.2.8 IPv4 (Internet Protocol version 4): The fourth version of the Internet Protocol, using 32-bit addresses to identify devices on a network.

4.2.9 IPv6 (Internet Protocol version 6): The most recent version of the Internet Protocol, using 128-bit addresses to overcome IPv4 address exhaustion.

4.2.10 LAN (Local Area Network)

A network that connects computers within a limited area such as a residence, school, or office building.

4.2.11 Latency

The time it takes for a data packet to move from source to destination, affecting the speed and performance of a network.

4.2.12 Network Topology

The arrangement of different elements (links, nodes) of a computer network, including physical and logical aspects.

4.2.13 Packet

A unit of data transmitted over a network, containing both control information and user data.

4.2.14 Protocol
A set of rules governing the exchange or transmission of data between devices.

4.2.15 TCP/IP (Transmission Control Protocol/Internet Protocol)
A suite of communication protocols used to interconnect network devices on the internet.

4.2.16 Throughput: The actual rate at which data is successfully transferred over a communication channel.

4.2.17 UDP (User Datagram Protocol)
A communication protocol that offers a limited amount of service when messages are exchanged between computers in a network that uses the Internet Protocol.

4.2.18 VLAN (Virtual Local Area Network)
A logical grouping of network devices that appear to be on the same LAN regardless of their physical location.

4.2.19 VPN (Virtual Private Network)
Creates a secure network connection over a public network such as the internet.

5. Protocol Description

The Authenticated Encrypted Relay Network (AERN) is an advanced cryptographic tunneling protocol explicitly designed to provide secure, authenticated, and anonymous communication channels between network devices. Unlike public anonymizing networks such as TOR, which utilize volunteer-operated nodes and layered asymmetric cryptography at each hop, AERN employs a fully private infrastructure with authenticated nodes. This infrastructure utilizes highly efficient symmetric cryptography for regular packet encryption, significantly improving performance and latency.

The protocol incorporates quantum-resistant cryptographic primitives (Kyber, McEliece, Dilithium, SPHINCS+), ensuring long-term security against emerging quantum computing threats. Robust certificate management administered via a root authority (ARS) and a centralized domain controller (ADC) further enhances security by tightly controlling node authentication, eliminating the risks associated with malicious or compromised nodes.

Note: parameter sets are chosen for 256-bits of post quantum security, these are changeable but the defaults are aggressive parameter sets for each asymmetric primitive set, ex. ML-KEM-1024 and SPHINCS-SHAKE-256.

5.1 Objectives

The **primary security and operational objectives** of the AERN protocol are explicitly designed to address known limitations of public anonymizing networks, notably **TOR**. Specifically, AERN aims to:

1. **Establish Authenticated and Secure Communication Channels:**
   Leverage quantum-resistant asymmetric cryptography exclusively during device registration and session initialization, thereafter using efficient symmetric cryptography (RCS cipher) to encrypt data exchanges, significantly reducing computational overhead compared to TOR's layered asymmetric encryption approach.
2. **Provide Robust Anonymity Against Metadata Harvesting:**
   Employ a fully private, authenticated node architecture with dynamic routing, standardized packet sizes, and per-packet route randomization to effectively prevent traffic correlation and metadata analysis attacks that have compromised TOR in practice.
3. **Ensure Long-Term Cryptographic Security:**
   Use quantum-resistant cryptographic algorithms (Kyber, McEliece, Dilithium, SPHINCS+) to secure the protocol against both current cryptographic attacks and anticipated future threats from quantum computing, addressing vulnerabilities not mitigated in classical anonymizing networks.
4. **Offer Scalability and Adaptability:**
   Allow straightforward deployment in diverse environments such as IoT networks, enterprise settings, or critical infrastructure, through streamlined management of authenticated nodes and computationally efficient encryption methods.
5. **Mitigate Known Cryptographic Attacks:**
   Implement strong cryptographic protections against classical attacks such as replay, man-in-the-middle (MITM), and key compromise through rigorous certificate validation, timestamp and sequence verification, and comprehensive authenticated encryption methods.

5.2 Key Components and Their Roles

AERN Network Infrastructure Components clearly delineate responsibilities to enhance network security and operational efficiency, explicitly addressing vulnerabilities prevalent in open networks such as TOR. These components include:

1. **AERN Root Security (ARS):**
   A secure, isolated trust anchor responsible for signing and managing device certificates, thereby eliminating the risk of unauthorized or malicious nodes commonly found in public anonymizing systems.
2. **AERN Domain Controller (ADC):**
   Centralized node responsible for secure network management, including device registration, authentication, certificate validation, topological synchronization, and revocation handling, significantly reducing the risk of compromise compared to TOR's distributed directory authorities.
3. **AERN Client Device (ACD):**
   Authenticated endpoint initiating encrypted communication through an entry node, ensuring secure and anonymous access to network resources.
4. **AERN Proxy Server (APS):**
   Authenticated, securely administered server nodes forming a fully meshed, encrypted communication network. Unlike TOR nodes, APS nodes authenticate all communication and operate exclusively within a secure, controlled infrastructure, significantly reducing vulnerability to node compromise.

5.2.1 ARS (AERN Root Security)

Role: Acts as the certificate authority for the network.

Functions:

- Generates and manages the root certificate (trust anchor).
- Signs device certificates to verify identity and authenticity.
- Can connect to the domain controller enabling a certificate signing proxy function.

5.2.2 ADC (AERN Domain Controller)

Role: Manages device registration and certificate validation.

Functions:

- Generates a certificate and stores the secret signing key.
- The certificate is signed by the root security server.
- Validates device certificates against the root server certificate.
- Maintains a master list of trusted devices (network topology).
- Distributes certificates and updates to devices.
- Manages device certificate revocation and resignation.
- Forwards updates on proxy nodes added or removed from the proxy mesh.

5.2.3 ACD (AERN Client Device)

Role: An end-user network device that initiates secure communication with a remote client device through the AERN proxy network.

Functions:

- Generates a certificate and stores the secret signing key.
- The certificate is signed by the root security server, directly or by proxy through the domain controller.
- Registers on the AERN network, receiving a list of proxy nodes in the mesh.
- Selects a random proxy node and connects to it to initiate a session through the proxy network, connecting to a remote client or server device.

5.2.4 APS (AERN Proxy Server)

Role: Central server in a proxy mesh, provides a logical network map as an entry or exit node, or as a proxy node facilitates secure communications between AERN Client Devices.

Functions:

- Generates a certificate and stores the secret signing key.
- The certificate is signed by the root server, directly or by proxy through the domain controller.
- Synchronizes with other proxy nodes, exchanging topological data and shared secrets.
- Communicates with the domain controller, receiving updates on proxy nodes and network control message broadcasts.
- Act as a nodal point on a proxy chain, receiving and decrypting, encrypting and forwarding message streams between proxy nodes and to end point devices.
- Can be an entry node, forwarding node, or exit node on the proxy chain.

5.3 Network Initialization

5.3.1 Root Server Initialization

Root Certificate Generation:

- The ARS generates its signature key-pair (public/private keys).
- Creates a public root certificate containing its signature verification key, serial number, issuer, configuration set, version, and expiration period.
- Securely stores the private key used for signing.

5.3.2 Domain Controller Initialization

Controller Certificate Generation:

- The ADC generates its signature key-pair.

- Creates a public certificate and stores the secret signing key.
- The ARS signs the ADC's certificate, establishing it as a trusted entity.

Network Management:

- The ADC begins managing device registrations and maintaining the network topology.

5.3.3 Proxy Server Initialization

Certificate Generation and Signing:

- Each device generates its own signature key-pair and certificate.
- Certificates are signed by the ARS directly or by proxy via the ADC.
- The ARS signs each device's certificate, establishing trust.

Registration with ADC:

- Devices register with the ADC, which validates their certificates.
- Devices are added to the network topology maintained by the ADC.
- Devices build partial copies of the topology, with knowledge of only the devices with which they interact.

5.3.5 Client Integration

Client Integration:

- The Client joins the network by registering with the ADC.
- Receives a list of available APS proxy servers.

5.4 Network Initialization

AERN network devices are initialized in a sequence:

1. ARS – Trust anchor
2. ADC – Network management
3. APS – Proxy servers
4. Clients – End user device

The root security server (ARS) signs the certificate of each device, either directly or once the domain controller (ADC) is initialized, through the ADC proxy signing feature.
Each device generates its own asymmetric signature verification/signing keypair.
The public signature verification key is a member of the certificate that each device generates independently. The generation of certificates and signing keys are the sole responsibility of the device itself, and only the originating device has knowledge of the secret signing key.
The device master encryption key (***mek***), is shared between proxy server devices, and is expanded by a key derivation function (SHAKE) which creates keys and nonces used to key two symmetric cipher instances, corresponding to the send and receive channels of an encrypted bi-directional tunnel.

When the certificate expiration time is exceeded, the master encryption key becomes invalid and a new certificate and master encryption key must be exchanged. In this case, the proxy announces the invalid state of the certificate to the domain controller, which broadcasts a revocation request to the network. The device with the expired certificate must then rejoin the network with an updated certificate. The re-registered APS then connects with each proxy in the network and performs an asymmetric key exchange with the proxy node, exchanging new master encryption keys, expanding that key and initializing the send and receive channel symmetric cipher instances, and re-establishing network synchronization.
The maximum expiration time set in a certificate SHALL NOT exceed the root certificate expiration time.
When a certificate is signed by the root, the certificate is hashed, and the hash is signed by the root signing key. The root signed hash is added to the child certificate, as well as the root certificate serial number.
If the user defined expiration time exceeds that of the root, the expiration time is set to the root's expiration time. No device certificate can have an expiration time that exceeds the root certificate's expiration time. Once the root certificate has expired, a new root certificate must be created and distributed to each device on the network, and every device on the network must renew their certificates, and rejoin the network.
Each exchange in AERN, whether it is a network message, part of the key exchange, or traffic on the encrypted tunnel, all of these functions use a packet valid-time feature. This adds the UTC time in seconds, a low-resolution time value to the packet header at the point of packet creation. If the time in the packet valid-time parameter received by the remote host exceeds the packet valid-time field by the packet time threshold (60 seconds by default), the message is deemed invalid and the circuit is torn down. The 22-byte outer tunnel header remains visible to the adjacent tunnel peer and is authenticated as associated data by the tunnel encryption function. The route map, relay payload header, session identifiers, fragment metadata, return flag, and carried payload are encrypted inside the relay plaintext. A receiving device selects the correct connection-set symmetric cipher instances from the authenticated peer connection context.
The packet creation timestamp and packet sequence number are added to the signature hash on network messages, where the packet message is hashed along with the valid-time timestamp and the packet sequence number, then signed by the devices asymmetric signing key.
During the tunnelling phase, the sequence number and packet creation time (***st***) are added to the ***additional data*** function of the symmetric cipher MAC used to encrypt and authenticate messages in the encrypted tunnel (the AEAD authenticated stream cipher RCS). In this way, message replay attacks are strongly mitigated, and all AERN messaging is protected from attack schemes that use packet header tampering, message alteration, or re-transmission of packet data.

5.4.1 Root certificate creation

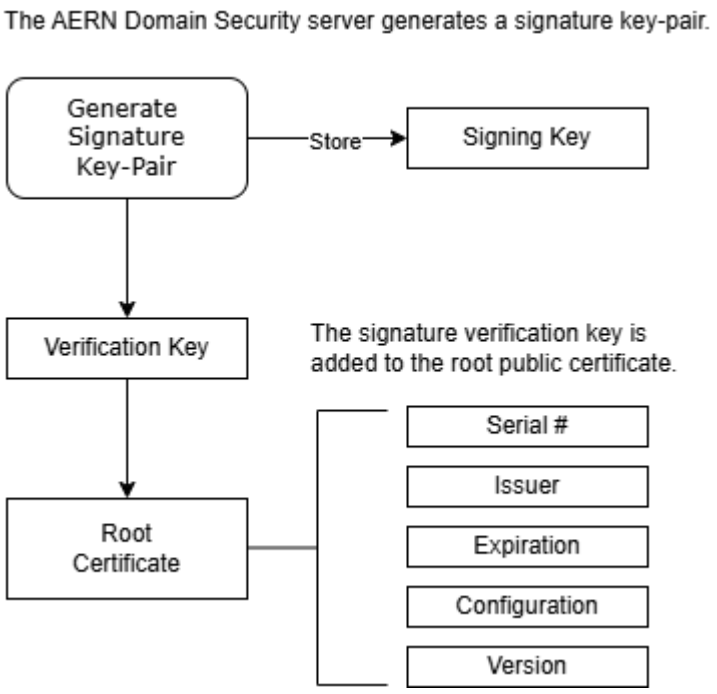


Figure 5.4.1 Root certificate generation.

The ARS generates a signature key-pair, stores the secret signing key, and adds the public signature verification key to the root certificate. The root certificate is made up of the following fields:

- The **signature verification key**, used to verify a root signature.
- The **issuer string**, identifies the certificate identity and formal name.
- The **serial number**, a unique 128-bit string used to identify the certificate.
- The **expiration time**, the valid *to* and *from* times, the time period in seconds during which the certificate is valid.
- The **configuration set** name, identifies the cryptographic primitives used by the key exchange from a set.
- The **version number**, the AERN protocol version number.

Expiration Time Structure

The expiration time structure holds the to and from valid times as seconds from the epoch.

Field	Size	Type	Description
from	64 bits	Uint64	The starting time in seconds.
to	64 bits	Uint64	The expiration time in seconds.

Table 5.4.1a: The aern_expiration_time structure.

Root Certificate

The root certificate is the network trust anchor used to validate the domain controller and device certificate signatures:

Field	Size	Type	Description
chash	32 bytes	Uint8 Array	Serialized root-certificate self-hash prefix. This field is present in the serialized root certificate and is verified on decode; it is not stored as a separate member of aern_root_certificate.
verkey	Variable	Uint8 Array	The serialized public verification key.
issuer	128 bytes	Uint8 Array	The certificate issuer.
serial	16 bytes	Uint8 Array	The certificate serial number.
expiration	16 bytes	Uint8 Array	The from and to certificate expiration times.
algorithm	1 byte	enum	The algorithm configuration identifier.
version	1 byte	enum	The certificate version.

Table 5.4.1b: The aern_root_certificate structure.

The serial number and issuer fields identify the certificate and the originating device.
The expiration time is the starting time and expiration time of the certificate in UTC seconds from the epoch. All certificates signed by the root, expire when the root expires.
The algorithm set name identifies which cryptographic set is used in the implementation, this can be the combination of asymmetric cipher and signature scheme families; Kyber-Dilithium, McEliece-Dilithium, and McEliece-SPHINCS+, further subdivided by the parameter sets used by each cipher and signature scheme.
The version number ensures that local and remote versions are synchronized.
The ARS root certificate is distributed to every device on the network and installed during device initialization, cached by those devices and used to authenticate certificates signed by the root security server.

5.4.2 ADC Initialization

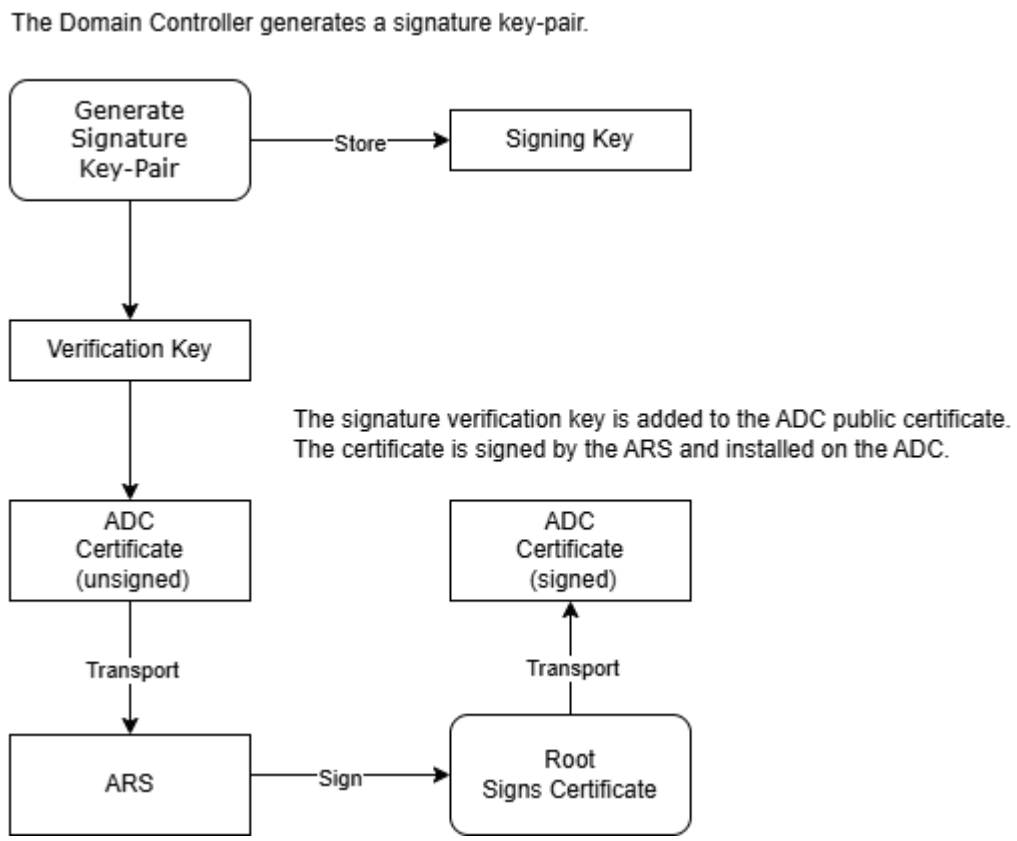


Figure 5.4.2a ADC certificate initialization

The ADC and all other child certificates have two additional parameters to the root certificate, the signature parameter which holds a copy of the ARS signed hash of the child certificate, and the root certificate serial number parameter.

Child certificate parameters:

- The **certificate signature**, generated by hashing the certificate, and signing the hash with the ARS signature key.
- The **root serial number** of the ARS server that signed this certificate.
- The **signature verification key**, used to verify a message signed by the corresponding device signing key.
- The **issuer string**, identifies the certificate's origin identity and formal readable network name.
- The **serial number**, a unique 128-bit string used to identify the certificate.
- The **expiration time**, the valid *to* and *from* times, the time period during which the certificate is valid.
- The **configuration set** name, identifies the cryptographic primitives used by the key exchange from a set.
- The **version number**, the AERN protocol version number.

Child Certificate

The child certificate is assigned to every device on the network, proxy servers, domain controller, and client devices:

Field	Size	Type	Description
csig	Variable	Uint8 Array	The certificate root-signed hash and signature bundle.
verkey	Variable	Uint8 Array	The serialized public verification key.
issuer	128 bytes	Uint8 Array	The certificate issuer.
serial	16 bytes	Uint8 Array	The certificate serial number.
rootser	16 bytes	Uint8 Array	The root certificate serial number.
expiration	16 bytes	Uint8 Array	The from and to certificate expiration times.
designation	1 byte	enum	The certificate type designation.
algorithm	1 byte	enum	The algorithm configuration identifier.
version	1 byte	Uint8	The certificate version.

Table 5.4.1c: The aern_child_certificate structure.

Once the ADC certificate has been signed by the ARS server, the ADC server can be brought online and is ready to handle registration requests and other administrative duties.

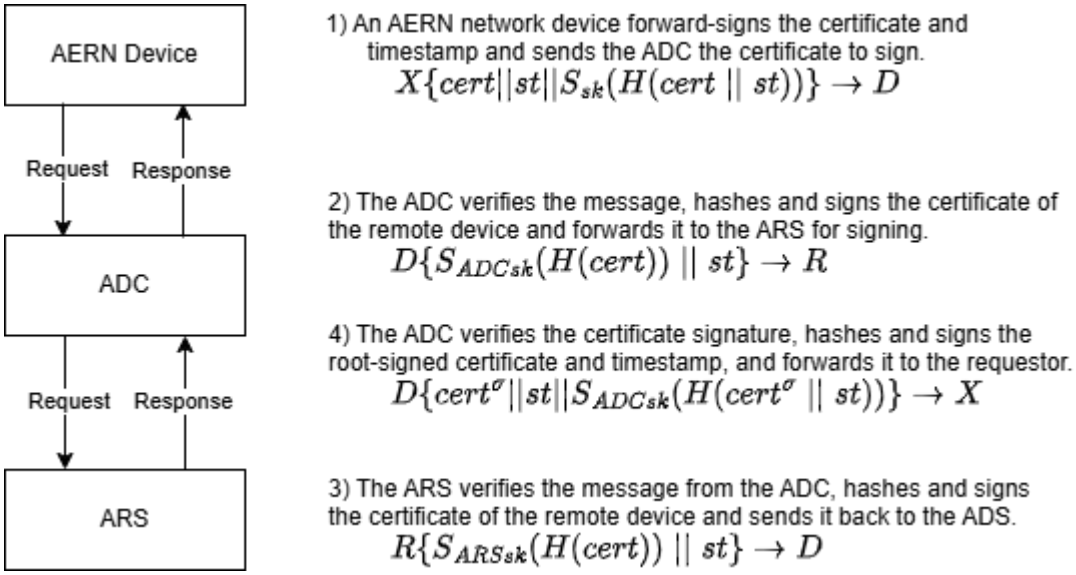


Figure 5.4.2b ADC proxy signing

The ADC certificate can be loaded onto the ARS to enable the proxy signing feature.
The ARS server is deliberately isolated, it has only one message capability, and this is to remotely sign a certificate as requested *only* by the ADC server. The ADC can act as a proxy for the signing of device certificates, allowing the isolation of the root server from other network devices (i.e. a 'militarized zone' on a secure private VLAN). The ARS stores the ADC certificate, and can only accept signing requests that have been issued and signed by the ADC. A network device sends a ***certificate signing request*** to the ADC, which forwards the certificate to the ARS, which signs the certificate, sends it back to the ADC, which forwards the certificate back to the requesting device.

5.4.4 APS Initialization

The proxy server sends a proxy ***registration request*** to the domain controller. The request contains the proxy server's root-signed certificate. The domain controller extracts the topological node information, and adds the proxy server to the domain controller's topological database, and then sends its root-signed certificate back to the proxy server along with a copy of the topological database, and a hash of the database forward-signed by the domain controller.
Nodes are arranged in the topological database of each device by ascending serial number rank, ensuring that all topological databases across all proxy servers and the domain controller are identical and synchronized. Connection lists, containing structures with pointers to the connection states and socket states, are arranged in the same order. Route hints refer to this shared topology ordering. In the current relay design, the route hints are contained in the encrypted route map serialized inside the relay plaintext and SHALL NOT appear as clear packet-header fields.

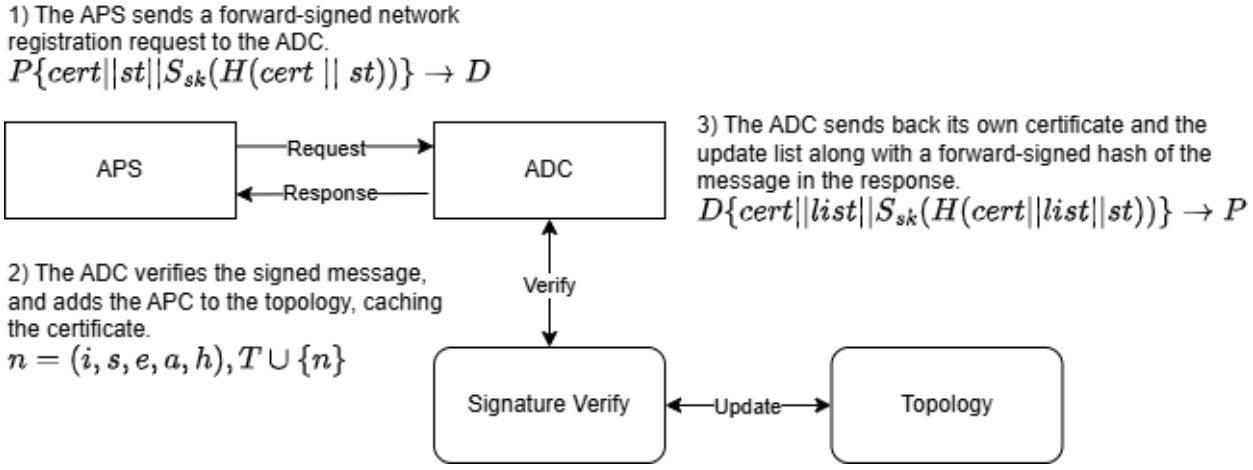


Figure 5.4.4a APS network registration.

Topological Node

The topological node structure contains information about a network device that is used to connect to, and verify that device:

Field	Size	Type	Description
address	22 bytes	Uint8 Array	The remote host's IP address.
chash	32 bytes	Uint8 Array	The hash of the nodes certificate.
issuer	128 bytes	Uint8 Array	The node's certificate issuer.
serial	16 bytes	Uint8 Array	The node's certificate serial number.
expiration	16 bytes	Uint8 Array	The from and to certificate expiration times.
designation	1 byte	enum	The device topological designation serialized as AERN_CERTIFICATE_DESIGNATION_SIZE.

Table 5.4.4a: The aern_topological_node structure.

Topological List

The topological list is an indexed array that contains a set of topological node structures:

Field	Size	Type	Description
topology	Variable	Uint8 Array	Pointer to the serialized topology array.
count	4 bytes	Uint32	The active node count in the topology list.

Table 5.4.4b: The aern_topological_list structure.

AERN stores a 64-bit monotonic topology version directly in aern_topology_list_state. Version zero is reserved for uninitialized or disposed topology state. Topology hashes are computed by aern_topology_hash() when required; they are not stored as a persistent topology-list hash member.

The proxy server adds the topological nodes in the update to its topological list. The new proxy server contacts each of the proxy servers in the ADC topological update list, and exchanges certificates. The certificate' signatures are verified, and the certificate is hashed and compared to the signature hash. The certificates are stored on the APS, and certificates for each proxy server are verified against the APS topological database sent by the domain controller by comparing the node certificate hash, expiration time, serial number, and issuer, with the corresponding fields in the topological node entry sent by the domain controller at registration.

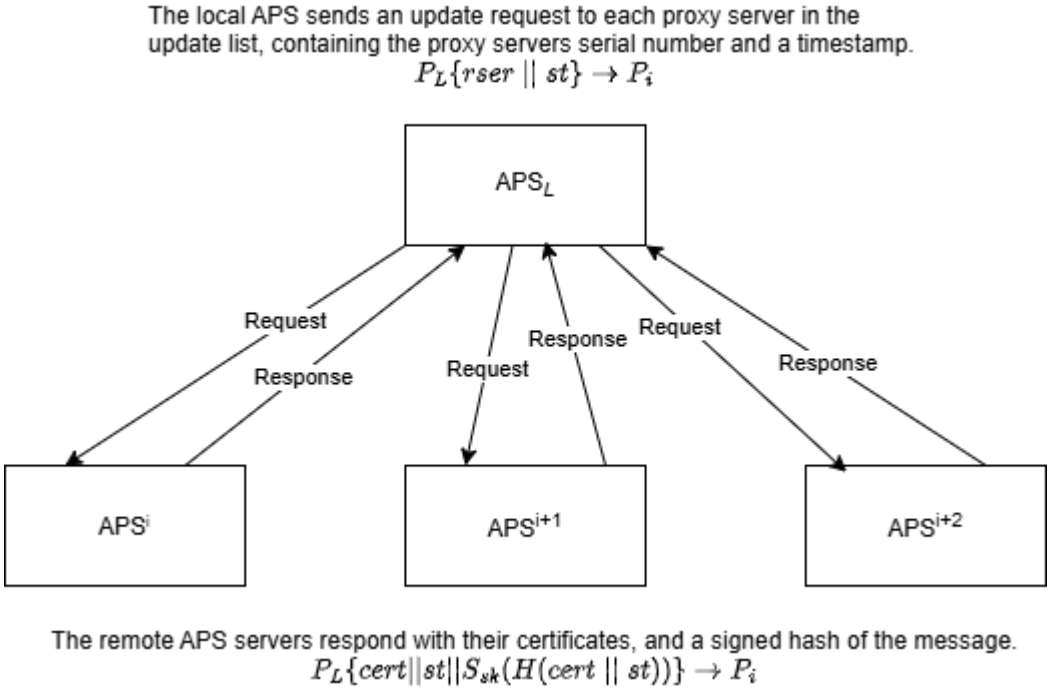


Figure 5.4.4b APS proxy update.

The APS then sends each proxy server a forward-signed ***incremental update request***, requesting the certificate of each device in the topological map it has received from the domain controller. The

certificate is verified against the corresponding node entry in the proxy server's topological database, and a validation of the root signature, and if passing authentication, the certificate is cached on the device (on authentication *failure* the device contacts the domain controller, which initiates a network convergence broadcast). The APS sends its own certificate in the request, which is verified by the remote server using the root signature. The APS server is added to the remote devices topological database and the certificate is cached.

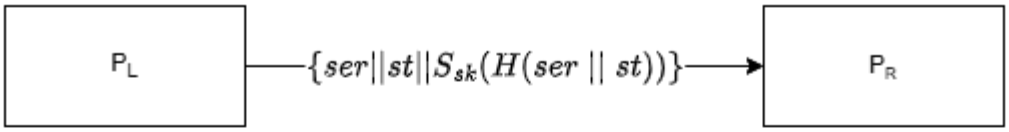
Once certificates have been exchanged with every server in the proxy mesh, the APS initiates a ***master encryption key exchange***; exchanging a shared secret with each proxy server on the network, expanding that key using a key derivation function to create the keys and nonces to initialize send and receive symmetric cipher instances for every proxy on the mesh, the virtual tunnels.

The APS generates an asymmetric cipher key-pair and timestamp, signs the public key and timestamp, and sends it to the remote proxy server.

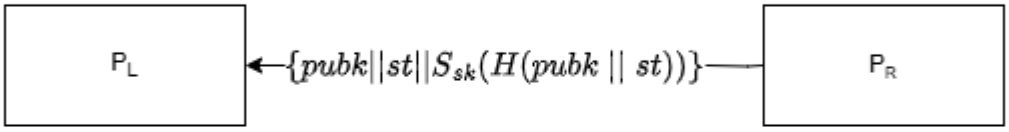
The proxy server verifies the signed key and timestamp and encapsulates a shared secret, hashes the ciphertext along with a timestamp and signs the hash. The signed ciphertext is sent back to the new proxy server, which verifies the signed hash and timestamp, and decapsulates the shared secret.

This master encryption key is expanded using a key derivation function (SHAKE) that creates the keys and nonces for the send and receive channel ciphers (RCS); two cipher instances are initialized one for each channel, and are stored in memory and associated with the proxy servers that share those keys. These cipher instances are *ad hoc* virtual encrypted tunnel interfaces, when a proxy server needs to send an encrypted message to another proxy server, it selects the pair of stream cipher instances associated with that remote proxy and encrypts or decrypts the packet payload with the associated cipher instance.

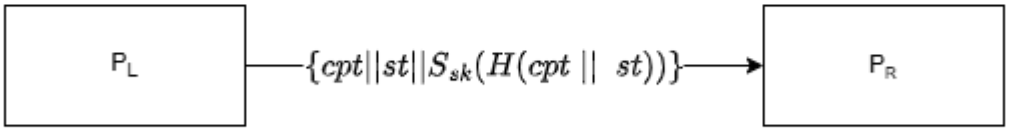
The local APS sends the remote proxy server its certificate serial number, and the signed hash of the serial number and timestamp.



The remote proxy server authenticates the message signature, generates a hash of the message and validates the message. The remote proxy server generates an asymmetric cipher key pair, hashes the public key and the timestamp, signs the message, and sends the message back to the APS.



The local APS verifies the certificate and the message signature, creates a secret master encryption key, encapsulates it using the public cipher key, hashes the ciphertext and timestamp, and sends the message to the remote proxy server.



The remote proxy authenticates the message signature, generates a hash of the message and validates the message. The remote proxy decapsulates the shared secret, and if the key exchange succeeds, sends a key synchronized message to the APS.

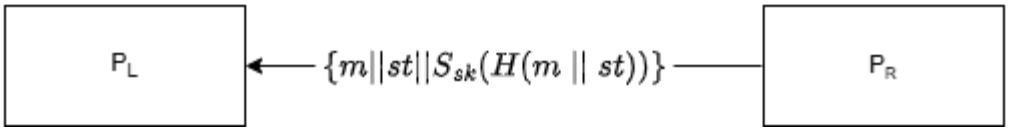


Figure 5.4.4c APS to proxy master key exchange.

Once every proxy shares a set of initialized cipher instances, a certificate, and topological node entry for every other proxy server on the network, the AERN network is considered ***synchronized*** and ready to accept client connections.

Note: a timestamp is used in every type of exchange in AERN, this is a low resolution (seconds) timestamp that accompanies various message exchanges and is checked against a threshold value that accounts for network delay or incorrect clock synchronization (+/- 60 seconds by default). If the received value falls outside of that threshold the exchange is terminated, an error condition is set, and the circuit collapsed. All servers within the AERN domain, must synchronize with an NTP server to provide reliable base system times.

Connection State

The connection state contains the send and receive symmetric cipher instances and information about the encrypted tunnel interface. This includes transmit and receive sequence members for strict tunnel sequencing and an authentication-failure counter used by the tunnel failure policy.

Name	Size	Type	Description
target	variable	qsc_socket	The remote node socket instance.

rxcpr	variable	struct	The receive-channel cipher state.
txcpr	variable	struct	The transmit-channel cipher state.
rxseq	8 bytes	uint64	The next expected receive sequence number.
txseq	8 bytes	uint64	The next transmit sequence number.
instance	4 bytes	uint32	The connection-state instance number.
authfail	4 bytes	uint32	The authentication-failure counter.
exflag	4 bytes	enum	The connection state flag.

Table 5.4.4c: The current aern_connection_state structure.

Each proxy in the mesh can communicate with every other proxy through an encrypted tunnel interface. These encrypted interfaces are considered to be logically at a layer beneath the packet transfer mechanism; they remain in memory, and any communication between proxy nodes is encrypted and decrypted by the tunnel interfaces, independent of individual client message streams. A proxy domain can contain a large number of proxy servers, the only real constraint is memory, as proxy inter-connections require that a pair of symmetric cipher states be resident in memory. The RCS cipher used in AERN has a state footprint of about 2 kilobytes per pairing, this is primarily the set of round keys and nonce associated with the cipher. Additional elements like the network buffers and cached function codes can add more to the memory profile, but on a well-tuned server the total should not represent more than 4-8 kilobytes per connection. Given modern server memory capacity, a server should be able to peer to thousands and potentially hundreds of thousands of other proxy servers in a very large network. Though, this is somewhat impractical and given that a network of this size could handle billions of simultaneous connections, it is unlikely that a domain of this scale would ever be required.

A more relevant limitation is the maximum hops value; when a route is chosen, a random number of hops is selected, a route map is constructed, with nodes along the path also being chosen at random by the entry node which creates the route-map. For real time protocols such as VoIP and video messaging, some limitation in the number of hops must be set. Proxy nodes use symmetric cryptography to encrypt traffic traveling between proxy nodes, the ciphers are pre-keyed and reside in memory, so traffic encryption is very fast, and unlike TOR which uses a computationally expensive asymmetric/symmetric hybrid encryption scheme across 3 hops, with AERN many hops can be added to a network path before noticeable delays causing signal jitter become problematic.

The default setting of the tunable constant AERN_ROUTE_MAXIMUM_HOPS is 16, which sets the maximum range of the random number of hops selected when composing a route map to 16, and the route can consist of any number of hops between the minimum AERN_ROUTE_MINIMUM_HOPS which is 3, and the maximum possible value. Testing a given client application on the proxy network on a live network, is the best way to determine the maximum allowed hop value, given the applications susceptibility to delay, but a value of 16 is an overall good general value.

Once every proxy has exchanged master encryption keys with every other proxy on the network, and initialized send and receive cipher instances, the network is considered to be synchronized.

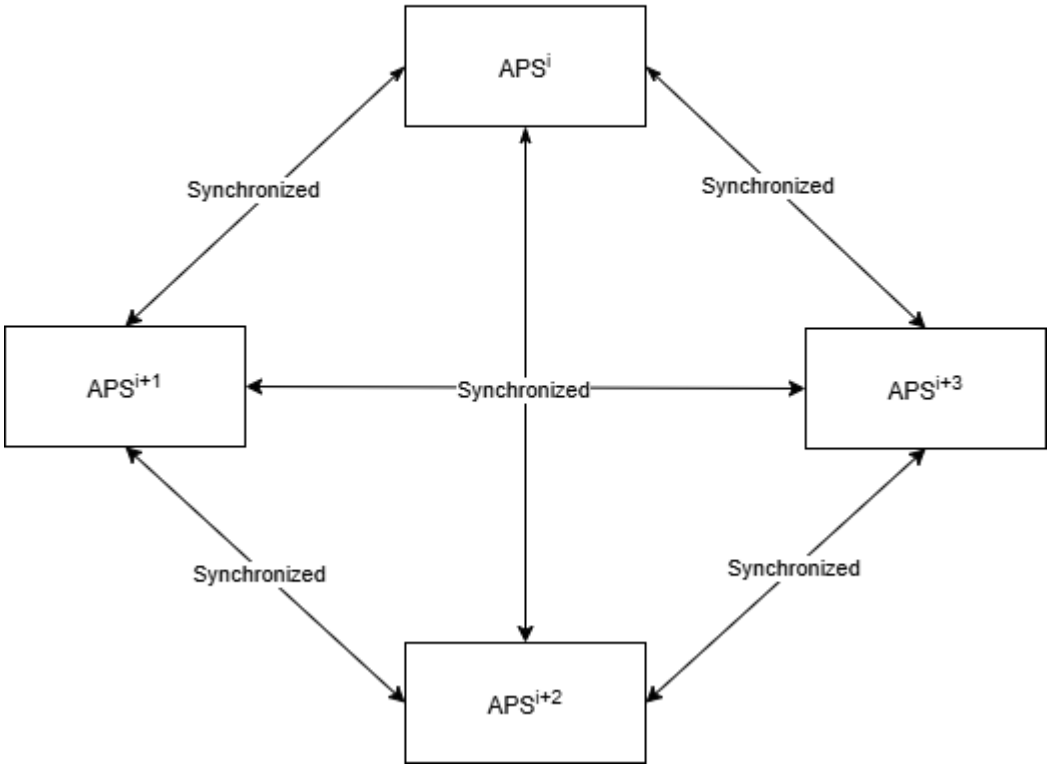


Figure 5.4.4e synchronized full-mesh proxy network.

5.4.5 Client Initialization

A client first registers through a web portal, authenticates, and generates a certificate. This certificate is signed by the root server through the domain controllers signing proxy service, and stored on the client along with the root public certificate.

When a client connects to an AERN domain for the first time, it undergoes a registration process, where the client sends a client registration request with a copy of its root signed certificate, created during the application initialization and software registration process, and a forward signed message consisting of the client's topological node information, which is hashed along with a timestamp and signed by the client's asymmetric signing key. The registration request is sent to the domain controller, which verifies the root signature of the client's certificate, and then uses the client's certificate to verify the message signature. The ADC then hashes the message and compares this with the signed hash of the message to complete authentication.

The domain controller assembles a list of proxy topological node entries for every active proxy server in the domain, organizes this list sorted by ascending serial number and then hashes this list along with the packet timestamp, and signs the hash with the server's asymmetric signing key. The proxy

node set along with the signed hash are sent back to the client, which installs the nodal information into the client's topological database, and stores the list hash.

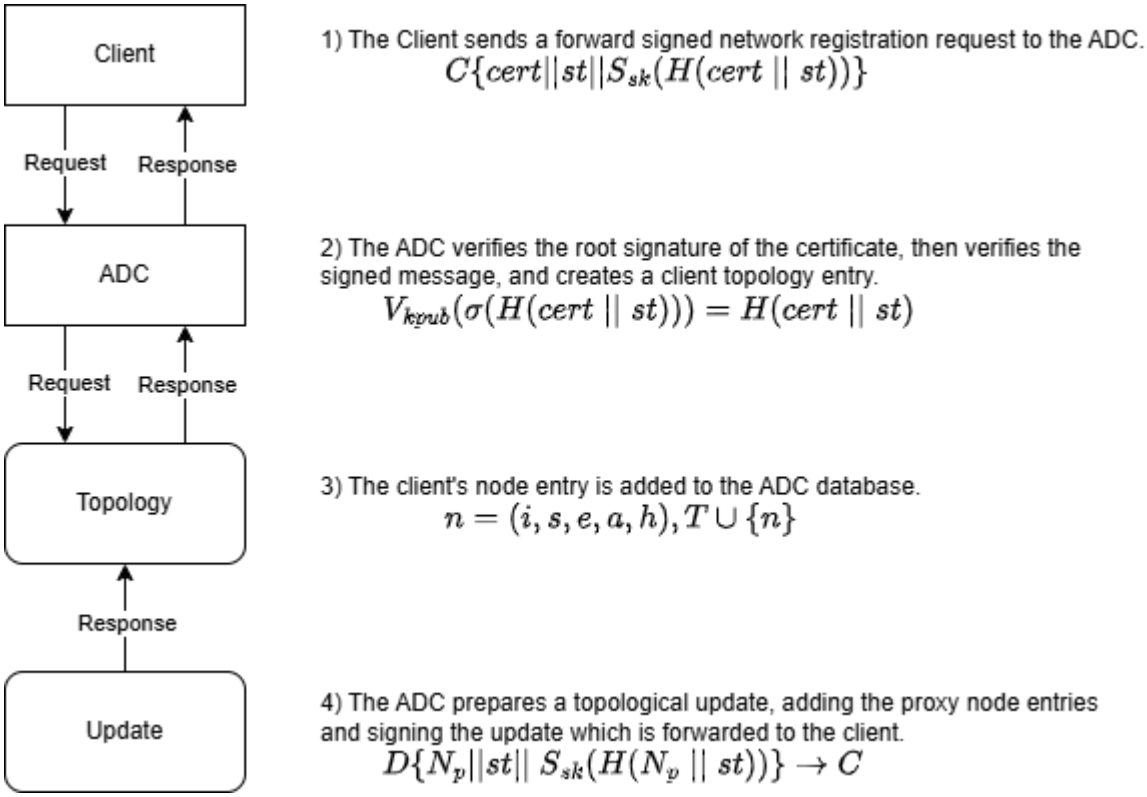
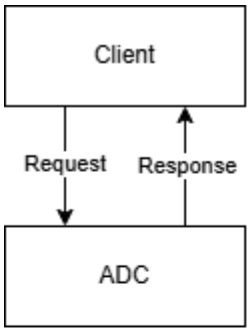


Figure 5.4.5a client domain registration request.

The client caches the node information for the proxy servers, this includes the IP address, server issuer name, certificate hash, serial number, and the certificate expiration time.

Note: The domain controller does *not* cache client certificates, but does store their topology information.

The first time a client connects to a domain controller the client sends a registration request, but if the domain controller is known to the client, the client sends a ***join request***. A join request queries the domain controller for changes to the topology, proxy servers that have been added or removed from the domain. The server creates the list of proxy servers organized by serial number and creates a hash of the nodal database; this *list hash* is cached on the domain controller. The domain controller signs this hash along with the packet timestamp and sends the message back to the client. The client compares the proxy list hash with its own cached list hash, and if they match, the client proceeds to selecting the entry node proxy, if this list does not match, the client flushes the previous cache and requests the new cache from the domain controller.

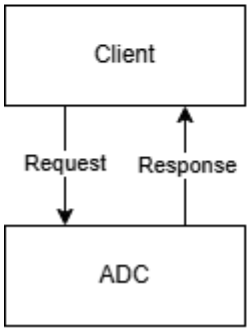


1) The Client sends a forward signed network join request to the ADC.

$$C\{cert||st||S_{sk}(H(cert \mid\mid st))\} \rightarrow D$$

2) The ADC verifies the root signature of the client certificate, then verifies the signed message, and forwards the topology list hash to the client.

$$D\{S_{sk}(H(lhash \mid\mid st))\} \rightarrow C$$



3) If the lists do not match, the client sends a topology request to the domain controller, requesting the new proxy list.

$$C\{cert||st||S_{sk}(H(cert \mid\mid st))\} \rightarrow D$$

4) The ADC verifies the root signature of the client certificate, then verifies the signed message, and forwards the topology list to the client.

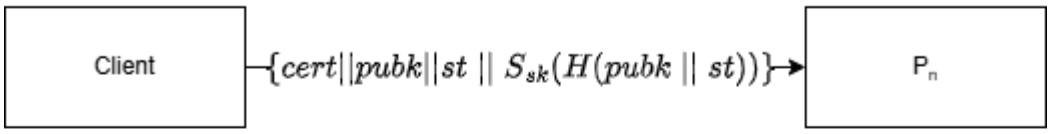
$$D\{N_p||st||S_{sk}(H(N_p \mid\mid st))\} \rightarrow C$$

Figure 5.4.5b client domain join request.

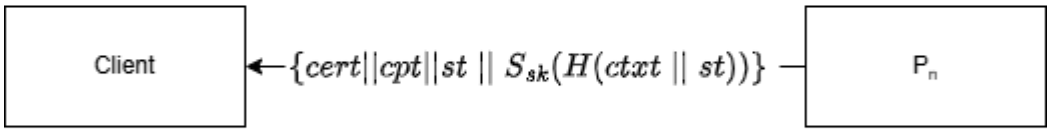
5.4.6 Client Entry Node Selection and Connection

Once the client has registered and joined the network, it selects an entry node; this node is chosen at random from the topological list of proxy servers it has received from the domain controller. Once the proxy node is selected, the client initiates an asymmetric key exchange with the proxy server and establishes an encrypted tunnel that persists through the duration of the session.

The client generates an asymmetric cipher key-pair. It hashes the public key along with the packet timestamp, it forward signs the hash with its asymmetric signing key. The client sends its certificate, the public key, and the signed hash of the public key and timestamp to the proxy server.



The proxy server authenticates the client certificates root signature and authenticates the message signature, generates a hash of the message and validates the message.
The proxy uses the public key to create the ciphertext and shared secret. It signs the ciphertext and packet timestamp, adds the signed hash, ciphertext, and its certificate to the packet.
The server expands the shared secret to produce the send and receive channel symmetric cipher keys, and keys the symmetric cipher instances, raising the send and receive channel interfaces.



The client verifies the proxy certificates root signature, then verifies the message signature using the proxies certificate, hashes the message and compares it to the signed hash to authenticate the message. The client extracts the shared secret from the ciphertext using the private asymmetric cipher key. The client expands the secret, generating cipher keys for the send and receive channel symmetric cipher instances. The client then raises the send and receive channels. The client sends a message to the proxy server signalling that the tunnel is in the ACTIVE state.

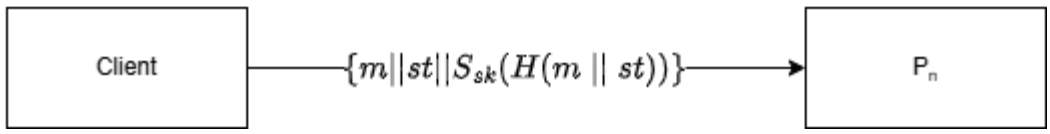


Figure 5.4.6a client to proxy key exchange.

Note: Proxy servers do not cache client certificates or topological node entries, but clients can cache proxy server certificates. If a client has cached the proxy server's certificate through a previous encrypted session, a hybrid of this key exchange takes place, wherein the proxy server does not send its certificate in the exchange. The client signals the possession of the entry nodes certificate in the connection request.

The client initiates a key exchange with the proxy server selected as the entry node. The client generates the asymmetric cipher key-pair and caches the private key. The client hashes the public key along with the packet timestamp and signs the hash with the asymmetric signing key, and sends the ***connection request*** to the proxy entry node containing the client's public certificate, the public cipher key, and the signed hash.

The proxy server authenticates the root signature of the client certificate, then authenticates the client signature of the hash of the public cipher key and timestamp using the client certificates signature verification key, it hashes the packet timestamp and public cipher key and compares the hash to the signed hash to complete authentication.

The proxy server uses the public cipher key to generate the shared secret and ciphertext. It hashes the ciphertext, and the ***connection response*** packet timestamp and signs the hash using its asymmetric signing key. The server adds its public certificate, the ciphertext, and the signed hash to the connection response packet, and sends it to the client. The server uses a key derivation function (SHAKE) to expand the shared secret, creating the symmetric cipher (RCS) keys and nonces for the send and receive channels of the encrypted tunnel. The server initializes both cipher instances, raising the send and receive interfaces of the tunnel to the client.

The client receives the connection response packet and verifies the root signature of the proxy server's certificate, if verified the certificate is cached for future sessions. The client uses the proxy server's certificate to verify the signature of the hash, then creates a hash of the ciphertext and connection response packet's timestamp and compares it to the signed hash to complete authentication. The client uses the private asymmetric cipher-key to decrypt the ciphertext and extract the shared secret. The client uses a key derivation function (SHAKE) to expand the secret into two symmetric cipher keys and nonces, and keys the cipher instances for the send and receive interfaces of the tunnel. The tunnel state is now considered ACTIVE and ready to transmit data. The client sends a ***connection established*** packet to the proxy server to verify the tunnel is now in the active state.

5.4.7 Route Map Creation

Once the client and proxy entry node have established an encrypted tunnel, the entry node is in the ready state. When the client submits a packet for relay transport, the ingress APS constructs a compact encrypted route path for the proxy mesh. The route path identifies the ordered set of proxy hops that SHALL be used for the packet, beginning with the ingress APS and terminating at the egress APS selected for the logical relay session. The serialized route path is carried only inside the encrypted relay plaintext and is never transmitted as a clear relay header field.

The egress APS is selected when the ingress creates the relay session. The ingress and egress hints are retained only for the lifetime of the session. Intermediate route hints are recomputed for each relay packet. This preserves the session relation required for reassembly and return traffic while preventing a long-lived fixed interior circuit. The compact route path is regenerated for each forwarded packet and does not expose route length, route cursor state, or the egress hint as cleartext metadata.

The route map is not a clear packet-header field. The route path is serialized inside the encrypted relay plaintext after a two-byte length prefix. A proxy receiving a packet first authenticates and decrypts the per-hop tunnel packet, then parses the encrypted route path to determine whether it is an intermediate hop or the terminal APS for the relay packet. The serialized route path is a consumed-next-hop structure. The receiving APS consumes the first nonzero future-hop entry, rewrites that entry to zero, reserializes the route path, and forwards the packet to the resolved next APS. If no future-hop entry remains, the receiving APS is terminal for the current relay direction.

Each route hint is an eight-bit, one-based APS topology ordinal into the topology-sorted proxy list; zero is reserved as the route terminator and as the empty value for unused future-hop slots. The topology list is ordered identically by all synchronized APS nodes under the ADC topology policy. A route hint therefore does not identify a network address directly on the wire; it identifies the common

index from which the receiving node resolves the next APS connection state. Because one byte is used for the serialized route hint, a single active route-addressable APS set is limited to 255 APS entries. Larger deployments MAY partition the relay domain, use topology shards, or define a future extended-hint profile, but the active compact route format described in this specification uses one-byte hints.

The serialized route-map state contains a sixteen-byte path array. The first byte is the origin hint for the route and the remaining fifteen bytes are ordered future-hop hints. The egress APS is represented as the final nonzero future-hop entry in the path. A zero future-hop entry terminates the route and all unused entries SHALL be zero. The in-memory route map also contains a generation-only hop-count field, but the hop count is not serialized as transmitted trust material. Route generation SHALL avoid selecting the same proxy as two consecutive hops. A proxy MAY appear more than once in a non-consecutive position when the route generator selects it independently.

Field	Size	Type	Description
path	16 bytes	uint8[16]	Sixteen one-byte route hints serialized as one-based APS topology ordinals. path[0] is the route origin. path[1] through path[15] are ordered future-hop entries. Zero terminates the remaining path and marks unused entries.
hop_count	1 byte local state	uint8	Generation-only hop count in the in-memory route map. It is not serialized as a transmitted trust value.

Table 5.4.7a: The aern_route_map structure.

The route path is serialized into the plaintext area that is subsequently encrypted and authenticated by the sender-to-receiver tunnel. Consequently, an observer outside the APS tunnel sees only the fixed-size relay packet and cannot read the route hints or terminal egress hint. A compromised intermediate APS can read the route path for the packet after decryption at that hop, but it remains bound to the authenticated tunnel packet and cannot be modified without causing authentication failure on the next hop. The consumed-path representation also avoids a mutable clear route cursor; forwarding state is represented only by the removal of the next future-hop entry inside the encrypted route path.

5.4.8 Network Traversal and Relay Packet Format

AERN relay traversal is implemented as a fixed-size per-hop encrypted packet carried over pre-established APS tunnels. The outer packet size is the network MTU value of 1500 bytes. The outer AERN packet header is 22 bytes and identifies the packet flag, message length, sequence number, and UTC creation time. The remaining 1478 bytes are the authenticated ciphertext region generated by the tunnel cipher. The decrypted relay plaintext is 1446 bytes. The relay plaintext contains a two-

byte length prefix, a sixteen-byte encrypted route path, a thirty-two-byte encrypted relay payload header, and the encrypted relay body and padding.

In the current relay design the route path, relay payload header, fragment identifiers, session identifiers, payload type, reserved relay byte, and return flag are inside the encrypted relay plaintext. Only the minimal per-hop tunnel header remains outside the encrypted region. That header is authenticated by the relay encryption function and is used for tunnel sequencing and freshness checking.

Layer	Size	Visibility	Description
Outer AERN packet	1500 bytes	Wire-visible length only	Fixed MTU-sized packet sent over the APS tunnel.
Outer packet header	22 bytes	Visible but authenticated	Contains tunnel flag, ciphertext length, sequence number, and UTC timestamp.
Ciphertext region	1478 bytes	Opaque	Authenticated encrypted tunnel payload.
Relay plaintext	1446 bytes after decryption	Visible only to decrypting APS	Contains length prefix, encrypted route path, relay payload header, and relay body.
Length prefix	2 bytes	Encrypted	Used relay payload length following the route path, measured as relay payload header plus body.
Route path	16 bytes	Encrypted	Consumed-next-hop route path containing origin and future one-byte APS hints.
Relay payload header	32 bytes	Encrypted	Session, packet, fragment, payload-type, reserved byte, and control flags.

Table 5.4.8a: Layered AERN relay packet format.

The relay packet format separates the tunnel layer from the relay session layer. The tunnel layer authenticates and decrypts a packet between two adjacent APS nodes. The relay session layer identifies the end-to-end ingress-to-egress session and the carried serialized transport packet. A forwarding APS SHALL NOT interpret the relay body as application data unless the route map identifies that APS as the terminal node for the current direction.

5.4.8.1 Encrypted Relay Payload Header

The relay payload header is serialized inside the encrypted relay plaintext. It binds the packet to a logical session and gives the terminal APS enough information to process control messages, reassemble fragments, and distinguish outbound and return traffic. All values in the relay payload

header are authenticated by the tunnel packet and are never transmitted as clear relay metadata. AERN uses a generic opaque data payload type rather than separate TCP, UDP, or ICMP relay payload types. The byte following the payload type is reserved and SHALL be set to zero unless a future protocol profile defines explicit protocol classification.

Field	Size	Type	Description
sessionid	8 bytes	uint64	Logical relay session identifier generated by the ingress APS.
packetid	8 bytes	uint64	Packet identifier used for fragment grouping and replay-resistant local processing.
fragseq	4 bytes	uint32	Fragment sequence number. Zero denotes an unfragmented packet.
fragcount	4 bytes	uint32	Total fragment count. A value of zero denotes an unfragmented packet.
msglen	4 bytes	uint32	Number of payload-body bytes carried in this fragment or unfragmented packet.
payloadtype	1 byte	enum	Relay payload type, including session-open, session-open-ack, session-close, opaque data, dummy, and error.
reserved	1 byte	uint8	Reserved for future protocol profiles. This field SHALL be set to zero in this protocol profile.
flags	2 bytes	uint16	Relay control flags. The return flag identifies egress-to-ingress packets.

Table 5.4.8b: The aern_relay_payload_header structure.

The following relay payload types are defined by AERN: session-open, session-open-ack, session-close, opaque data packet, dummy packet, and error. Data payloads carry serialized packet bytes whose interpretation is performed by the configured backend transport boundary, not by a clear protocol byte in the relay payload header. Dummy packets SHALL use the dummy payload type, SHALL set the reserved byte to zero, and SHALL NOT be delivered to a backend transport.

5.4.8.2 Ingress-to-Egress Relay Session Establishment

AERN uses an explicit encrypted relay session-open protocol between the ingress APS and the egress APS. The first outbound data packet for a new flow causes the ingress APS to allocate a relay session entry in the pending state. The ingress then creates an encrypted session-open payload containing the session identifier, destination address, one-byte ingress hint, one-byte egress hint, destination port, a reserved byte, and flags. The serialized session-open payload is 36 bytes. The reserved byte SHALL be zero unless a future protocol profile assigns it a meaning.

Session-open field	Size	Description
sessionid	8 bytes	Logical session identifier.
destination	22 bytes	Serialized destination address.
ingresshint	1 byte	One-based route hint for the ingress APS.
egresshint	1 byte	One-based route hint for the egress APS.
port	2 bytes	Destination port.
reserved	1 byte	Reserved for future use and set to zero.
flags	1 byte	Session-control flags.

Table 5.4.8c: The aern_relay_session_open structure.

When the egress APS receives and authenticates a session-open payload as the terminal relay node, it validates the destination, ingress hint, egress hint, reserved byte, flags, and session identifier. If the request is valid, the egress creates an active egress-side session cache entry and returns a session-open acknowledgement to the ingress APS. The acknowledgement contains the session identifier, status byte, flags byte, and a reserved sixteen-bit field.

Session-open-ack field	Size	Description
sessionid	8 bytes	Session identifier being acknowledged.
status	1 byte	Acknowledgement status. Zero denotes success.
flags	1 byte	Acknowledgement flags.
reserved	2 bytes	Reserved for future use and set to zero.

Table 5.4.8d: The aern_relay_session_open_ack structure.

Packets received from the client while a session is pending are placed into a pending queue. When the ingress receives a valid acknowledgement from the egress, the ingress marks the session active and releases queued packets into the relay path. If the acknowledgement does not arrive before the configured pending timeout, queued packets are discarded and the session-open attempt fails.

5.4.8.3 Backend Transport Interface

AERN uses a backend-neutral callback interface at both egress and ingress boundaries. After the egress APS authenticates, decrypts, validates, and reassembles a relay packet, it submits the serialized transport packet to the configured egress transport callback. The callback MAY bind AERN to a TUN or TAP device, a raw socket backend, a userspace TCP or UDP stack, an application-layer forwarder, or another local transport adapter.

The backend transport interface is outside the cryptographic relay core. The relay core is responsible for session binding, route traversal, packet authentication, fragmentation, reassembly, return-flag processing, and fixed-size relay packet construction. The backend is responsible for interacting with the external destination or local host interface. A conforming implementation SHALL NOT expose relay metadata through the backend interface.

Return traffic follows the inverse logical session path. The egress backend delivers a serialized packet to the AERN return interface. The egress APS wraps the packet in a relay payload, sets the return flag, fragments it if required, builds a route back toward the ingress hint, and sends it through the encrypted APS mesh. The ingress terminal path validates the return flag and session state, reassembles fragments, and delivers the resulting serialized packet to the ingress transport callback.

5.4.8.4 Relay Fragmentation and Reassembly

Fragmentation is performed at the encrypted relay payload layer, not in the clear outer packet header. If a serialized transport packet exceeds the relay data payload capacity, the ingress or egress divides the packet into fragments. Each fragment carries the same session identifier and packet identifier, and contains a fragment sequence number, fragment count, message length, and fragment body. The fragment metadata is inside the encrypted relay payload header. AERN provides AERN_MAX_USER_PAYLOAD = 1428 bytes inside the encrypted relay content region after the length prefix and sixteen-byte route path. After the 32-byte relay payload header is deducted, AERN_RELAY_DATA_PAYLOAD_SIZE and AERN_FRAG_CHUNK_SIZE are 1396 bytes. The maximum fragment count is 4096, subject to implementation memory limits and the fragment-cache memory policy.

A terminal APS accepts fragments only after the per-hop tunnel packet has authenticated and decrypted successfully. Fragment reassembly is keyed by session identifier, packet identifier, direction, and relay payload type. Relay payload fragments may arrive out of order at the fragment cache layer, while the per-hop tunnel sequence remains strict. Incomplete fragment cache entries expire after the configured fragment-cache timeout of 30,000 milliseconds.

The implementation bounds the maximum fragment count by the configured maximum fragment constant. A fragment set is valid only when every fragment is within the allowed count, every fragment length is within the relay data-payload capacity, the reserved relay header byte and payload type are consistent, and the reconstructed message length matches the accumulated fragment bytes. Failure of any condition SHALL discard the fragment set and SHALL NOT deliver data to the backend.

5.4.8.5 Dummy Relay Traffic and Ingress Delay

AERN includes optional APS dummy relay traffic. Dummy packets use the same fixed 1500-byte packet size and the same tunnel encryption mechanism as ordinary relay packets. A dummy relay payload contains random session and packet identifiers, the dummy payload type, a reserved relay header byte set to zero, and random body bytes. A terminal APS that receives a valid dummy packet SHALL discard it silently and SHALL NOT deliver it to the backend transport.

Dummy traffic is governed by local utilization policy. Dummy generation is enabled by default, uses a 10 percent utilization floor and a 25 percent utilization ceiling, attempts generation at randomized intervals between 50 and 250 milliseconds, evaluates a 1,000 millisecond traffic accounting window,

and limits dummy emission to eight packets per accounting window. The local target window is based on 128 MTU-sized packets and is not transmitted or logged.

The implementation also supports optional randomized ingress delay. When enabled, outbound client packets may be delayed by a random amount between zero and 25 milliseconds before entering the APS mesh. This delay is a local timing-obfuscation control. It reduces deterministic linkage between client ingress events and relay injection events, but it is not equivalent to mixnet batching and SHALL NOT be represented as a complete defense against a fully global timing adversary.

5.4.9 Cipher Re-instantiation, Replay Protection, and Forward Security

Each APS tunnel maintains independent transmit and receive cipher states. Tunnel packets include an authenticated sequence number and UTC creation time. The current relay policy uses strict next-sequence validation for ordered APS-to-APS transport. The configured replay-window size is zero, indicating that out-of-order tunnel packets are not accepted at the per-hop tunnel layer. Relay payload fragments remain reorderable only after the tunnel packet has authenticated and decrypted successfully.

A receiving tunnel endpoint validates the packet flag, ciphertext length, sequence number, and timestamp before accepting the decrypted relay plaintext. A packet outside the valid time threshold, a packet with an unexpected sequence number, or a packet with an invalid authentication tag is rejected. The connection state includes an authentication-failure counter. Repeated authentication failures beyond the configured relay limit may invalidate or tear down the peer tunnel according to local policy.

Forward-security policy is described in terms of tunnel reinitialization and key-refresh capability. When tunnel reinitialization is performed, both peers SHALL derive fresh symmetric state from authenticated key material and SHALL erase superseded key-dependent state.

Field	Size	Type	Description
target	variable	qsc_socket	Remote node socket state associated with the tunnel.
rxcpr	variable	RCS state	Receive-channel symmetric cipher state.
txcpr	variable	RCS state	Transmit-channel symmetric cipher state.
rxseq	8 bytes	uint64	Next expected receive-channel sequence number.
txseq	8 bytes	uint64	Next transmit-channel sequence number.
instance	4 bytes	uint32	Connection-state instance identifier.
authfail	4 bytes	uint32	Authentication-failure counter for the peer tunnel.

exflag	4 bytes	enum	Connection state flag.

Table 5.4.9a: The current aern_connection_state structure.

5.4.10 Disconnects, Errors, and Session Teardown

A relay session may terminate by an explicit session-close payload, by backend failure, by tunnel failure, by pending session-open timeout, by idle session timeout, or by administrative removal of a proxy from the topology. Session teardown erases the ingress and egress session-cache entries, clears pending and delay queues associated with the session, and releases any incomplete fragment sets.

A graceful client disconnect causes the ingress APS to send a session-close payload through a newly generated route to the egress APS. The payload is fixed-size at the relay layer and carries no backend data. The egress APS tears down the egress-side session and releases backend state. If the client exits without sending a close payload, the ingress relies on the configured idle timeout. The lifecycle layer uses a thirty-minute client idle timeout, while the relay session layer uses its own shorter cache and pending-open timers.

A forwarding APS that cannot reach an intermediate next hop may attempt the configured hop-bypass logic where the route map permits it and where the failed hop is not the terminal ingress or egress node. A failed peer is reported to the ADC. The ADC may perform keepalive checks and, if the peer remains unavailable, broadcast a revocation or topology update. Proxy traffic-flow metadata SHALL NOT be written to proxy logs. Administrative error reporting SHALL be restricted to device identity, event time, and error class, and SHALL NOT include client content or external destination data.

6. Mathematical Description

AERN uses various messages between devices to accomplish network tasks.
The domain controller handles network control messaging, including certificate revocation, network convergence, network join requests, registration and resignation messages.
Messages are also passed between clients and proxy servers, such as connection requests and key exchanges.
All messages are signed using the senders secret asymmetric signing key, and are verified by the receiving device using the senders' public certificate. This not only guarantees the authenticity of the sender, but a packet creation time and sequence number are included in the message hash that is signed by the originating device, protecting the message from replay attacks.
This section contains a list of message functions used by AERN, and their mathematical descriptions.

6.1 Converge Broadcast

Overview:

Network convergence is an administrative event called from the ADC. Each proxy server on the network is sent a copy of their topological node database entry. The serialized node entry for the remote device is hashed along with a timestamp and sequence number, and the hash is signed by the domain controller and sent to the device.

The signature is verified by the device using the ADC's public certificate, the local node entry is serialized and hashed, and compared with the signed hash. If the hashes match, the entry in the ADC topological database is *synchronized*, if the entries do not match, the device serializes the current topological database entry and the certificate, signs them with the current signature key, which is signed by the root security server (ARS), and sends it back to the ADC. The ADC verifies the new certificate using the ARS public certificate to verify the signed hash of the certificate embedded in the signature field. The old entry is purged, a new topological entry is added to the database, and the new certificate is stored.

* Note that the proper procedure after a certificate update on a APS, is to resign from the network, and then rejoin with a new certificate.

API:

- *aern_network_converge_request()*
- *aern_network_converge_response()*

Applies to:

- APS
- ADC

Mathematical Description:

Let:

- H_{TS}^{σ} be the signed hash of $H(T_D \parallel st)$ signed using D's private key.

- H be the hash function.
- K_{pri} be the secret signing key.
- K_{pub} be the signature verification key.
- *Sign* be the asymmetric signing function.
- *st* be the sequence number and valid-time timestamp.
- T_D be the topological node of device *D*.
- Verify be the asymmetric signature verification function.

The converge broadcast request:

The ADC creates the converge request using the remote device's topological node, hashed with the timestamp and signed.

$H_{TS}{}^{\sigma} = \text{Sign}_{adcKpri}(H(T_D \parallel st))$

$\text{Request}(T_D) = (T_D \parallel H_{TS}{}^{\sigma})$

The device verifies the ADC's signature.

$\text{Verify}_{adcKpub}(H_{TS}{}^{\sigma}) = H(T_D \parallel st)$

The converge response:

The responding device signs the response message, and sends it to the ADC.

$\sigma = \text{Sign}_{rapsKpri}(H(T_D \parallel st))$

$\text{Response}(T_R) = (T_R \parallel \sigma)$

Proof of Security:

Correctness: The response is only generated if the request is valid. Both the request and the response signatures are verified using the public key of the respective device.

Proof: The request signature is:

$\sigma(H(T_D \parallel st)) = \text{Sign}_{Kpri}(H(T_D \parallel st))$

Upon receiving the request, the recipient checks the validity of the signature using:

$\text{Verify}_{Kpub}(\sigma(H(T_D \parallel st))) = H(T_D \parallel st)$

If the signature verification passes, the recipient knows the request is authentic. The node structure sent by the ADC, containing information about the remote device including certificate serial number, issuer, and expiration *to* and *from* times, is verified by the receiving device. If the node values match, the receiver signs its serialized node structure along with the timestamp and sequence number, and sends it back to the ADC as confirmation that the topology is aligned. If the values do not match, or the authentication or message is invalid, the receiver sends back an error message. If the ADC

receives an error, or the connection times out, the remote node is removed from the ADC's topology, and the device's certificate is revoked, removing it from the topology list of every device on the network.

Integrity: The hash H(T_D || *st*) ensures that the certificate cannot be altered. Any tampering will result in a failed signature verification.

Replay Protection: A timestamp and sequence number are included in the hash H(T_D || *ts*) and checked to ensure that broadcasts cannot be reused maliciously.

6.2 Incremental Update

Overview:

The incremental update functions retrieve a devices certificate. When a device joins the network, the ADC sends a list of resources available for that device. When a APS joins the network the ADC sends it a list of available network proxy servers, when a Client joins the ADC sends a list of all proxy servers on the network.

The Client and APS synchronize with devices on the list sent by the ADC, creating a topological database. The topology is a local list containing information about resources that the device uses on the network. A topological node is an element in the list that contains important information like the nodes IP address, issuer name, expiration time, certificate hash and serial number. This information is used to connect to the device, request its certificate, verify the certificate, and interact with the device on the network.

The **aern_topology_node_state** structure defines the state information for a device within the AERN topology. This includes details like network address, certificate information, and the device's designation.

Once the device has obtained the certificate and added the node to its topology, the device can exchange a shared secret between devices using the master encryption key (*mek*) asymmetric key exchange.

During network registration, the Client and APS device receive a list of resources they will use on the network.
The Client or APS queries each node on this list, requesting the devices public certificate. The requestor uses the remote devices serial number S_D as the request message.

API:

- aern_network_incremental_update_request()
- aern_network_incremental_update_response()

Applies to:

- APS

Let:

- C_D^{σ} be the root signed certificate of device D.
- st be the sequence number and valid-time timestamp.
- σ be the asymmetric signature.
- H be the hash function.
- H_{CS}^{σ} be the signed certificate and timestamp hash.
- K_{pri} be the private signing key.
- K_{pub} be the signature verification key.
- ser_D be the requested certificate serial number.
- $Sign$ be the asymmetric signing function.
- st be the sequence number and packet creation timestamp.
- Verify be the asymmetric signature verification function.

The incremental update request:

The device sends an incremental update request with the remote device certificate serial number.

$\text{Request}(S_D) = (ser_D)$

The responding device sends the serialized certificate, and a hash of the certificate and the packet headers valid-time timestamp and sequence number, signed with its secret signing key.

The incremental update response:

$H_{CS}^{\sigma} = \text{Sign}_{respKpri}(H(C_D^{\sigma} \,||\, st))$

$\text{Response}(C_D^{\sigma}) = (C_D^{\sigma} \,\|\, H_{CS}^{\sigma})$

The certificate signature is verified and a hash of the certificate is compared to the signed hash, and the hash contained in the topological node entry. The certificate hash must match the hash stored in the node information sent by the ADC. If the certificate is validated, it is added to the devices certificate store.

$\text{Verify}_{respKpub}(H_{CS}^{\sigma}) = H(C_D^{\sigma} \,||\, st)$

Proof of Security:

Correctness: The response is only generated if the request is valid and the serial number in the request matches the responder's certificate serial number. The responder's certificate is verified by the requestor using the root public certificate. The response message signature is verified using the received public key of the respective device.

Proof: The response signature is:

$\sigma(H(C_D^{\sigma} \,||\, st)) = \text{Sign}_{Kpri}(H(C_D^{\sigma} \,||\, st))$

Upon receiving the request, the recipient checks the validity of the certificates' signature using:

$\text{Verify}_{rootKpub}(\sigma(H(C_D))) = H(C_D)$

The response message including the responder's certificate and valid-time timestamp are then verified using the validated responder's certificate.

$\text{Verify}_{devKpub}(\sigma(H(C_D^{\sigma} \parallel st))) = H(C_D^{\sigma} \parallel st)$

If the root signature verification passes, the certificate is authentic. The certificate is then used to authenticate that the message is valid and sent by the responding device. If any of these checks fail; root signature, responder message signature, hashes, sequence, packet creation valid-time, or the certificate hash comparison with the node hash value sent by the ADC, the certificate is rejected.

Integrity: The hash $H(C_D^{\sigma} \parallel st)$ ensures that the certificate cannot be altered. Any tampering will result in a failed signature verification.

Replay Protection: A timestamp and sequence number are included in the hash $H(C_D \parallel ts)$ and checked to ensure that the requests cannot be reused maliciously.

6.3 Master Encryption Key Exchange

Overview:

The master encryption key exchange is an authenticated asymmetric key exchange in which a shared secret is exchanged between devices. In the implementation, MEK is the tunnel-use name for the derived shared secret and directional cipher material; the underlying wire exchange remains the MFK exchange implemented by aern_network_mfk_exchange_request() and aern_network_mfk_exchange_response(). A Client and an APS server exchange a master encryption key to create an ephemeral session tunnel between the Client and the Entry node, and APS servers exchange master encryption keys with each other to create a fully meshed network of encrypted tunnels. Every device on the network receives the root server public certificate upon installation, which is used to authenticate certificates between devices during these exchanges.
Master encryption keys can be replaced periodically, triggered by a time threshold to further goals of forward secrecy. Traffic is temporarily queued (the key exchange takes approximately +/- 40 milliseconds), and the queue emptied when the connected proxy servers have been re-keyed and the tunnels are re-established. These re-keying thresholds are activated once the proxy server is enabled, and continue to refresh keying material during the operation lifetime of the APS server.

API:

- aern_mek_exchange_request() / aern_network_mfk_exchange_request()
- aern_mek_exchange_response() / aern_network_mfk_exchange_response()

Applies to:

- APS
- Client

Mathematical Description:

Let:

- C_D^{σ} be the root signed certificate of device *D*.
- *ct* be the asymmetric cipher-text.
- Enc be the asymmetric encapsulation function.
- Dec be the asymmetric decapsulation function.
- H be the hash function.
- H_{CS}^{σ} be the signed certificate and timestamp hash.
- H_{ES}^{σ} be the signed asymmetric ciphertext and timestamp hash.
- H_{PS}^{σ} be the signed public cipher key and timestamp hash.
- KGen be the asymmetric cipher key generation function.
- K_{pub} be the asymmetric signature public key.
- K_{pri} be the asymmetric signature private key.
- *mfk* be the master fragment key.
- *pk* be the asymmetric cipher public key.
- *sk* be the asymmetric cipher secret key.
- Sign is the asymmetric signing function.
- *ss* be the shared secret.
- Verify is the asymmetric verification function.

The requestor sends an exchange request to the device. The message contains the requestors serialized certificate, and a valid-time timestamp.

Note: APS servers and Clients retain ARS, APS, and ADC public certificates.

$H_{CS}^{\sigma} = \text{Sign}_{reqtKpri}(H(C_D^{\sigma} \parallel st))$

$\text{Request}(C_D^{\sigma}) = (C_D^{\sigma} \parallel H_{CS}^{\sigma})$

The responder verifies the certificates root signature.

$\text{Verify}_{rootKpub}(C_D^{\sigma}) = H(C_D)$

The responder validates the requestors certificate and the valid-time timestamp are then verified using the validated responder's certificate.

$\text{Verify}_{devKpub}(H_{CS}^{\sigma}) = H(C_D^{\sigma} \parallel st)$

The responder generates a keypair using the asymmetric cipher. It stores the private key, hashes and signs the public cipher key and valid-time timestamp, and sends it to the requestor.

$pk, sk = \text{KGen}(\lambda, r)$

$H_{PS}^{\sigma} = \text{Sign}_{respKpri}(H(pk \parallel st))$

$\text{Response}(pk) = (pk \parallel H_{PS}^{\sigma})$

The signed public key is sent to the requestor. The signature, hash, and timestamp are verified, and the requestor uses the public key to encapsulate a shared secret.

$\text{Verify}_{respKpub}(H_{PS}^{\sigma}) = H(pk \parallel st)$

$ct = \text{Enc}_{pk}(ss)$

The shared secret is retained by the requestor and is the *master encryption key*. The ciphertext is hashed along with the valid-time timestamp, and the hash is signed by the requestors signing key.

$H_{ES}^{\sigma} = \text{Sign}_{reqtKpri}(H(ct \parallel st))$

$\text{Request}(ct) = (ct \parallel H_{ES}^{\sigma})$

The responder verifies the message hash using the requestors public verification key, then compares the hash against the hashed ciphertext and timestamp.

$\text{Verify}_{devKpub}(H_{ES}^{\sigma}) = H(ct \parallel st)$

If the ciphertext is validated, the ciphertext is decrypted using the responders private cipher key.

$ss = \text{Dec}_{sk}(ct)$

Proof of Security:

Correctness: The key exchange consists of three steps:

- The requestor sends a signed hash of its certificate and timestamp to the responder.
- The responder signs a hash of the public cipher key and timestamp and sends it to the requestor.
- The requestor signs a copy of the ciphertext and timestamp and sends it to the responder.

Proof: Given the definition of digital signatures and the message m:

$\sigma(H(m \parallel st)) = \text{Sign}_{Kpri}(H(m \parallel st))$

The verification function computes:

$\text{Verify}_{Kpub}(\sigma(H(m \parallel st))) = H(m \parallel st)$

Since Verify_{Kpub} is the inverse of Sign_{Kpri}, the signature is valid if it was signed by the matching private key. The hash is generated from the message and compared to the signed hash for equality.

Integrity: Since $H(m \parallel st)$ is hashed and signed, any change to the certificate or signature would cause the verification to fail. The hash function used (e.g., SHAKE) is collision-resistant, ensuring that an attacker cannot forge C_D or $\sigma(H(m \parallel st))$.

Replay Protection: A timestamp and sequence number are included in the hash H(*m* || *ts*) and checked to ensure that broadcasts cannot be reused maliciously.

6.4 Register Update Request

Overview:

When a Client or APS registers with the ADC to join an AERN network. The ADC verifies the devices certificate, then sends a list of topological nodes that are available for that device, a copy of its own root-signed certificate, and adds the device to the topology.

API:

- *aern_network_register_update_request()*
- *aern_network_register_update_response()*

Applies to:

- Client
- ADC
- APS

Mathematical Description:

Let:

- C_D^{σ} be the root signed device certificate.
- H be the hash function.
- H_{CS}^{σ} be the signed certificate and timestamp hash.
- H_{CLS}^{σ} be the signed certificate, list, and timestamp hash.
- K_{pri} be the private asymmetric signing key.
- K_{pub} be the public asymmetric verification key.
- *list* be the list of nodes.
- Sign be the asymmetric signing function.
- σ be the signature.
- Verify be the asymmetric signature verification function.

The requestor sends a *register update request* to the ADC. The message contains the requestors serialized certificate, and a signed hash of the certificate and the valid-time timestamp.

The registration update request:

$H_{CS}^{\sigma} = \text{Sign}_{reqtKpri}(H(C_D^{\sigma} \,||\, st))$

$\text{Request}(C_D^{\sigma}) = (C_D^{\sigma} \,\|\, H_{CS}^{\sigma})$

The ADC responder validates the requestors certificate root signature and the valid-time timestamp are then verified using the validated responder's certificate.

$\text{Verify}_{rootKpub}(\sigma(H(C_D)) = H(C_D)$

$\text{Verify}_{devKpub}(H_{CS}^{\sigma}) = H(C_D^{\sigma} \,||\, st)$

The ADC generates a list of topological nodes for the device; APS servers and Clients receive a list of APS servers.
The ADC hashes and signs the list, its certificate, and valid-time timestamp and sends it to the APS or Client.

The registration update response:

$list = \{ D_1, D_2, \ldots D_n \}$ where D_i is a topological node.

$H_{CLS}^{\sigma} = \text{Sign}_{adcKpri}(H(C_D^{\sigma} \,||\, list \,||\, st))$

$\text{Response}(C_D^{\sigma} \,||\, list) = (C_D^{\sigma} \,||\, list \,\|\, H_{CLS}^{\sigma})$

The requestor verifies and stores the ADC certificate, generates a topological node for the ADC, and is registered on the network. The requestor adds the list of nodes to the topological list, and will synchronize certificates with each device using the *incremental update* function, and then exchange master encryption keys using the *master encryption key exchange*. Once the device has the certificate and master fragment key of each device, its topology is considered *synchronized*.

$\text{Verify}_{rootKpub}(\sigma(H(C_D)) = H(C_D)$

$\text{Verify}_{adcKpub}(H_{CLS}^{\sigma}) = H(C_D^{\sigma} \,||\, list \,||\, st)$

6.5 Remote Signing Request

Overview:

The root domain security server (ARS) only has a single networked function. Remote signing allows *only* the ADC to connect to the ARS, to act as a proxy for certificate signing. The ADC can sign certificates for devices on the network by connecting to the ARS, and forwarding the certificate to be signed. The ARS has a copy of the ADC certificate, allowing it to verify the signing request message.

API:

- *aern_network_remote_signing_request()*
- *aern_network_remote_signing_response()*

Applies to:

- ADC

- ARS

Mathematical Description:

Let:

- C_D be a device certificate.
- H be the hash function.
- H_{CS}^{σ} be the signed certificate and timestamp hash.
- K_{pri} be the private asymmetric signing key.
- K_{pub} be the public asymmetric verification key.
- Sign be the asymmetric signing function.
- σ be the signature.
- Verify be the asymmetric signature verification function.

The ADC sends a *remote signing request* to the ARS. The message contains the serialized certificate to be signed, and a signed hash of the certificate and the valid-time timestamp.

The remote signing request:

$H_{CS}^{\sigma} = \text{Sign}_{adcKpri}(H(C_D \parallel st))$

$\text{Request}(C_D) = (C_D \parallel H_{CS}^{\sigma})$

The ARS validates the ADC's remote signing request signature, the certificate hash, and the valid-time timestamp.

$\text{Verify}_{adcKpub}(H_{CS}^{\sigma}) = H(C_D \parallel st)$

The ARS signs the certificate, and sends it back to the ADC.

The remote signing response:

$C_D^{\sigma} = \text{Sign}_{rootKpri}(H(C_D))$

$\text{Response}(C_D^{\sigma})$

The ADC verifies the root signature, and can now forward the certificate to the network device.

$\text{Verify}_{rootKpub}(\sigma(H(C_D)) = H(C_D)$

6.6 Resign Request

Overview:

A Client or an APS can resign from the network by sending a resign request to the ADC. In the case of a proxy server the ADC sends out a revoke request broadcast removing the device's certificate and nodal information from every node on the network.

API:

- *aern_network_resign_request()*
- *aern_network_resign_response()*

Applies to:

- APS
- Client
- ADC

Mathematical Description:

Let:

- H be the hash function.
- H_{SS}^{σ} be the signed serial number and timestamp hash.
- K_{pri} be the private asymmetric signing key.
- K_{pub} be the public asymmetric verification key.
- *list* be the list of nodes.
- Sign be the asymmetric signing function.
- σ be the signature.
- Verify be the asymmetric signature verification function.

The requestor sends a *resign request* to the ADC. The message contains the requestors certificate serial number, and a signed hash of the serial number and the valid-time timestamp.

The resignation request:

$H_{SS}^{\sigma} = \text{Sign}_{devKpri}(H(S_D \,||\, st))$

$\text{Request}(S_D) = (S_D \,\|\, H_{SS}^{\sigma})$

The ADC looks up the serial number in its topology, loads the device certificate and validates the signed message.

$\text{Verify}_{devKpub}(H_{SS}^{\sigma}) = H(S_D \,||\, st)$

The requesting device erases its topology, and must make a register request to the ADC to rejoin the network. The ADC sends a revocation broadcast to a subset of relevant nodes on the network.

6.7 Revoke Broadcast

Overview:

The revocation request is a broadcast message that instructs nodes on the network that a certificate has been revoked, and that device is to be removed from the network. Network members that receive this message, disconnect and destroy associated state, delete the devices certificate and remove it from the local topological database. Revocation broadcasts are sent to proxy servers. Clients are informed of a revocation when they join the network by connecting to the ADC, before selecting an entry node on the network. In the case of a proxy server failure or administrative removal, all devices including Clients and APS servers are sent a revocation broadcast message.

API:

- *aern_network_revoke_broadcast()*
- *aern_network_revoke_response()*

Applies to:

- APS
- ADC

Mathematical Description:

Let:

- H be the hash function.
- H_{SS}^{σ} be the signed serial number and timestamp hash.
- K_{pri} be the private asymmetric signing key.
- K_{pub} be the public asymmetric verification key.
- *list* be the list of nodes.
- S_D be the device certificate serial number.
- Sign be the asymmetric signing function.
- *st* be the sequence number and valid-time timestamp.
- σ be the signature.
- Verify be the asymmetric signature verification function.

The revocation message contains a signed copy of the device certificate serial number to be revoked.

HSSσ = SignadcKpri(H(SD || st))

Request(S_D) = (S_D ‖ H_{SS}^{σ})

The ADC sends the revocation out to a list of devices.

L = { D_1, D_2, …, D_n }

For each $i \in L$ = Broadcast(L_i, (S_D ‖ σ))

6.8 Join Request

Overview:

Before a Client contacts an ADC server and begins a session, it contacts the ADC with a join request. The client sends the domain controller a signed copy of its topology list hash, a hash of the ordered topological nodes in its database. If this hash matches the ADC topological list hash, the ADC responds with a simple acknowledgment mirroring the same hash value, but if the hash differs from the domain controllers list hash, the ADC sends a new topological list to the Client. This function ensures that the client is synchronized with the master topological database kept by the domain controller, and that if proxy servers have been added or revoked, the client removes or adds the proxy servers and synchronizes its topological database.

API:

- *aern_network_join_request()*
- *aern_network_join_response()*

Applies to:

- Client
- ADC

Mathematical Description:

Let:

- C_D^{σ} be the root signed device certificate.
- H be the hash function.
- H_{CS}^{σ} be the signed certificate and timestamp hash.
- K_{pri} be the private asymmetric signing key.
- K_{pub} be the public asymmetric verification key.
- Sign be the asymmetric signing function.
- *thash* be the hash of the topological database.
- σ be the signature.
- Verify be the asymmetric signature verification function.

The Client requestor sends a *join request* to the domain controller. The message contains the Client's public certificate and a signed hash of the Client's topological database hash and the valid-time timestamp.

The join request:

$H_{CS}^{\sigma} = \text{Sign}_{devKpri}(H(thash \parallel st))$

$\text{Request}(C_D^{\sigma} \parallel H_{CS}^{\sigma})$

The ADC responder validates the requestors certificate and the valid-time timestamp are then verified using the validated responder's certificate. The ADC verifies that the hash sent by the requestor matches its own topological database hash.

$\text{Verify}_{rootKpub}(\sigma(H(C_D)) = H(C_D)$

$\text{Verify}_{devKpub}(H_{CS}^{\sigma}) = H(thash \parallel st)$

The registration response:

If the hash matches the ADC signs its copy of the topology hash, and sends this message to the Client.

$H_{CS}^{\sigma} = \text{Sign}_{adcKpri}(H(thash \parallel st))$

$\text{Response}(C_D^{\sigma} \parallel H_{CS}^{\sigma})$

If the topology hash does not match, the ADC creates a copy of the proxy server topology, hashes and signs this copy, and forwards this to the Client.

$list = \{ P_1, P_2, \ldots P_n \}$ where P_i is a topological node.

$H_{CLS}^{\sigma} = \text{Sign}_{adcKpri}(H(list \parallel st))$

$\text{Response}(C_D^{\sigma} \parallel list \parallel H_{CLS}^{\sigma})$

The Client verifies the ADC certificate, and parses the new list. Nodes that have been removed from the ADC list are removed from the Client's topological list. Proxy nodes that have been added to the ADC topology are integrated into the Client's topological database.

$\text{Verify}_{rootKpub}(\sigma(H(C_D)) = H(C_D)$

$\text{Verify}_{adcKpub}(H_{CS}^{\sigma}) = H(list \parallel st)$

6.9 Relay Packet Encryption

The encrypted relay packet is modeled as a composition of a per-hop tunnel packet and an encrypted relay payload. Let Hdr be the outer AERN tunnel header, let RM be the serialized sixteen-byte route path, let RH be the relay payload header, let B be the relay body, and let L be the two-byte encrypted length prefix. The plaintext input to the tunnel cipher is P = L || RM || RH || B || pad, where pad fills the remaining plaintext region with random or zeroized bytes according to the implementation policy. The value L encodes the used relay payload length following the route path, namely the length of RH || B, and does not include the outer tunnel header, the length prefix itself, or the serialized route path.

The sender computes C = Enc_Ktx(Hdr, P), where Enc denotes the authenticated tunnel encryption function and Hdr is included in the authenticated context. The receiver accepts P only if Dec_Krx(Hdr, C) verifies. Any change to Hdr, C, RM, RH, or B causes authentication failure. Because RM and RH are inside P, route hints, session identifiers, fragment identifiers, payload type,

reserved relay byte, and return flags are confidential against observers outside the decrypting tunnel endpoint.

Correctness follows from the tunnel encryption and decryption relation Dec_Krx(Hdr, Enc_Ktx(Hdr, P)) = P when the transmit and receive states are synchronized. Integrity follows from the unforgeability of the authenticated encryption tag. Replay resistance follows from the inclusion of the tunnel sequence number and timestamp in the authenticated header and from strict next-sequence validation.

6.10 Route Map Forwarding

Let T be the synchronized topology list, and let hint h identify T[h]. Let RM = (p0, p1, ..., p15) be the sixteen-byte consumed route path recovered from the decrypted relay plaintext, where p0 is the origin hint and p1 through p15 are ordered future-hop hints. The receiving APS determines whether it is terminal for the current direction by scanning the future-hop entries for the first nonzero hint. If no future-hop entry exists, the APS is terminal. If the packet is not terminal, the APS resolves the first nonzero future-hop hint against T, rewrites that consumed entry to zero, reserializes RM, encrypts the updated plaintext under the tunnel state shared with the next APS, and transmits the fixed-size relay packet.

A route path is valid only if every non-empty hint resolves to a valid APS node in the current topology, if the next-hop hint is not the receiving APS itself, and if consecutive duplicate hints are absent in the generated route. Because all route-path updates occur inside authenticated tunnel packets, an adversary cannot redirect a packet to an arbitrary topology index without either compromising a forwarding APS or forging the tunnel authentication. The one-byte hint encoding constrains the active route-addressable APS set to 255 entries; larger deployments require topology partitioning, sharding, or a future extended-hint profile.

6.11 Relay Session-Open State Machine

Let S be a relay session identifier generated by the ingress APS. The ingress state machine begins in state none. On the first outbound serialized transport packet, the ingress creates state pending(S), queues the packet, and transmits a session-open payload to the egress APS. The egress accepts the session-open payload only if the destination, ingress hint, egress hint, reserved byte, flags, and session identifier are valid. On acceptance, the egress creates active(S) and returns session-open-ack(S).

The ingress transitions from pending(S) to active(S) only after receiving a valid acknowledgement from the selected egress. Pending packets are released only after this transition. If acknowledgement fails or the pending timer expires, the ingress erases pending(S) and discards queued packets. This prevents unauthenticated data release and binds data delivery to an acknowledged egress-side session.

For return traffic, the egress constructs payloads with the return flag set. The ingress accepts return packets only for active sessions and only when the return flag, session identifier, reserved relay byte,

packet identifier, and fragment metadata are consistent. This creates direction separation between outbound and return packet flows while preserving a single logical relay session.

6.12 Relay Fragmentation

Let M be a serialized transport packet of length |M|. If |M| is at most the relay data payload capacity of 1396 bytes, M is carried as a single relay payload with fragseq = 0 and fragcount = 0. If |M| exceeds that capacity, M is partitioned into fragments M1, M2, ..., Mk, where k is bounded by the maximum fragment count of 4096 and each |Mi| is no greater than the relay data payload capacity. Each fragment carries the same session identifier and packet identifier.

The terminal APS reassembles M only after receiving all fragments 1 through k before the fragment cache timeout expires. The reconstructed value is accepted only if the total recovered length equals the declared message length and if all fragment headers agree on session, packet identifier, payload type, reserved relay byte, and direction flags. Otherwise, the fragment set is discarded.

6.13 Dummy Packet Generation

A dummy packet is a relay packet for which RH.payloadtype = dummy and RH.reserved = 0. The dummy body is filled with random bytes and is transmitted through the same per-hop encrypted tunnel and route-path forwarding procedure as ordinary packets. A terminal APS that authenticates and decrypts a dummy payload performs no backend delivery and erases the payload.

Let U be the local traffic utilization estimate over the accounting window. Dummy generation is permitted only when U is below the configured ceiling and is preferentially used when U is below the configured floor. The implementation bounds the number of generated dummy packets per window. This policy increases background indistinguishability during sparse traffic periods while avoiding unbounded bandwidth growth.

6.14 Replay and Freshness

Let seq_rx be the next expected tunnel receive sequence number. A received tunnel packet with sequence seq is accepted only if seq = seq_rx and the packet timestamp lies within the configured valid-time threshold. On acceptance, seq_rx is incremented. A duplicate packet has seq < seq_rx and is rejected. A reordered packet has seq > seq_rx and is rejected under the strict sequencing policy.

This strict per-hop rule differs from a sliding replay window. AERN uses ordered APS tunnel transport; therefore a replay window is unnecessary at the tunnel layer. Fragment reordering is handled after successful tunnel authentication by the relay fragment cache and does not weaken per-hop replay protection.

6.15 Backend Transport Boundary

The backend transport callback is not part of the cryptographic tunnel proof. The cryptographic relay core proves confidentiality, integrity, freshness, route-map authentication, session binding, and direction separation up to the point where a terminal APS delivers or receives a serialized transport packet. The security of the external transport destination, operating-system device, or application backend is a separate implementation boundary.

6.16 Announce Broadcast and Response

Overview:

The announce broadcast is an ADC-originated control message used to announce a device certificate or topology node to the relevant network population. The request is broadcast over the ADC-maintained topology list and is accepted only when the announced node and the ADC signature validate against the local trust state.

API:

aern_network_announce_broadcast()
aern_network_announce_response()

Applies to:
ADC
APS
Client

Mathematical Description:

Let TD be the topological node being announced, let CDσ be the corresponding root-signed certificate when carried or locally available, let st be the packet sequence and valid-time timestamp, let H be the hash function, and let SignadcKpri be the ADC signing operation.

HASσ = SignadcKpri(H(TD || st))

Broadcast(TD) = (TD || HASσ)

Each receiving device verifies VerifyadcKpub(HASσ) = H(TD || st), validates the timestamp and sequence number, and verifies that the announced certificate or node material is consistent with the ARS root and local topology policy. On success the node information is accepted or refreshed. On failure the message is rejected and no topology state is modified.

Proof of Security:

Correctness follows because an honest ADC signs exactly the announced topological node and freshness state. Integrity follows from the collision resistance of H and the unforgeability of the signature scheme. Replay resistance follows because st is inside the signed value and is checked against the protocol time threshold and sequence policy.

6.17 Converge Update

Overview:

The convergence-update function processes an APS-signed topology correction at the ADC. It is distinct from the ADC convergence request and response. A convergence update is accepted only after the APS certificate is verified against the ARS root and the correction is validated against the versioned topology state.

API:

aern_network_converge_update()

Applies to:
ADC
APS

Mathematical Description:

Let TD be the corrected topology node supplied by the APS, let CDσ be the APS certificate, let v be the incoming topology version, and let st be the sequence and valid-time timestamp.

HCUσ = SignapsKpri(H(CDσ || TD || v || st))

Update(CDσ, TD, v) = (CDσ || TD || v || HCUσ)

The ADC verifies VerifyrootKpub(CDσ), then verifies VerifyapsKpub(HCUσ) = H(CDσ || TD || v || st). If the certificate, topology node, version ordering, sequence, and timestamp are valid, the ADC updates the aern_topology_versioned_state and increments or accepts the monotonic topology version according to local policy.

Proof of Security:

An attacker cannot inject an accepted topology correction without either forging the APS signature or presenting a valid root-signed APS certificate. Version validation prevents stale topology state from replacing newer topology state.

6.18 Fragment-Key Request and Response

Overview:

The fkey request and response functions exchange or verify fragmentation-key material associated with a peer topology node. This is an administrative key-fragment operation in the active AERN network API. It is not the MPDC multi-party key-fragment collection protocol and SHALL NOT be interpreted as MPDC-style multi-party strengthening.

API:

aern_network_fkey_request()
aern_network_fkey_response()

Applies to:
APS

Mathematical Description:

Let MF be the local master fragment key, let tok be the exchange token, let F be the output fragment, let L be the local topological node, let R be the remote topological node, and let st be the sequence and valid-time timestamp.

F = KDF(MF || tok || L.serial || R.serial || st)

HFσ = SignKpri(H(F || tok || L || R || st))

Request(tok, L, R) = (tok || L || R || HFσ)

The responder validates the topology nodes, derives the corresponding fragment from its local MFK state, and returns an authenticated response bound to the same token and node identities.

Proof of Security:

The token and node identities bind the fragment to one exchange instance. A stale or replayed fragment is rejected because the signed or authenticated value includes st and the exchange token. A fragment derived for one peer pair cannot be substituted for another peer pair without causing verification failure.

6.19 Register Request and Response

Overview:

The register request and response functions perform the basic certificate-authenticated registration contact with the ADC. They exchange and authenticate the local device certificate and ADC certificate without necessarily transferring the full topology list. Register-update functions provide the topology-list variant.

API:

aern_network_register_request()
aern_network_register_response()

Applies to:
Client
APS
ADC

Mathematical Description:

Let CDσ be the registering device certificate, let CAσ be the ADC certificate, and let st be the packet sequence and valid-time timestamp.

HDRσ = SigndevKpri(H(CDσ || st))

Request(CDσ) = (CDσ || HDRσ)

The ADC verifies VerifyrootKpub(CDσ) and VerifydevKpub(HDRσ) = H(CDσ || st). If accepted, the ADC signs its response certificate and freshness state:

HARσ = SignadcKpri(H(CAσ || CDσ.serial || st))

Response(CAσ) = (CAσ || HARσ)

The registering device verifies the ARS signature over CAσ and verifies the ADC response signature. No topology state is accepted unless the certificate chain and freshness checks pass.

Proof of Security:

Mutual certificate authentication prevents an unauthenticated device from registering and prevents a non-ADC endpoint from impersonating the registration authority. Replay resistance follows from the timestamp and sequence number being signed.

6.20 Register Update Versioned Response

Overview:

The versioned register-update response is the current ADC response form used when the ADC maintains a versioned topology state. It wraps the register-update semantics with a monotonic topology version so that clients and APS nodes can reject stale topology material.

API:

aern_network_register_update_v2_response()

Applies to:
ADC
APS
Client

Mathematical Description:

Let L be the topology list, let v be the ADC topology version, let CAσ be the ADC certificate, and let st be the packet sequence and valid-time timestamp.

HLVσ = SignadcKpri(H(CAσ || L || v || st))

Response(CAσ, L, v) = (CAσ || L || v || HLVσ)

The receiver verifies the ADC certificate against the ARS root, verifies the ADC signature, and accepts the topology only if the version is not stale relative to its local aern_topology_versioned_state.

Proof of Security:

Topology authenticity follows from the ADC signature. Stale-list replacement is prevented by the monotonic version check. Integrity follows because any modification to the list or version changes the signed hash.

6.21 Topological Query Request and Response

Overview:

The topological query functions allow a device to query the ADC for a topology node by issuer or serial information. The response returns an authenticated topology node, allowing the requester to locate and validate a device without accepting unauthenticated directory data.

API:

aern_network_topological_query_request()
aern_network_topological_query_response()

Applies to:
Client
APS
ADC

Mathematical Description:

Let q be the issuer string or serial query, let SD be the requester serial number, let TD be the returned topology node, and let st be the sequence and valid-time timestamp.

$HQ\sigma = SigndevKpri(H(q \parallel SD \parallel st))$

$Request(q, SD) = (q \parallel SD \parallel HQ\sigma)$

The ADC verifies the requester certificate and signature, searches the topology list, and signs the result:

$HTQ\sigma = SignadcKpri(H(TD \parallel q \parallel SD \parallel st))$

$Response(TD) = (TD \parallel HTQ\sigma)$

The requester accepts TD only if $VerifyadcKpub(HTQ\sigma) = H(TD \parallel q \parallel SD \parallel st)$, the timestamp and sequence number are valid, and the returned node satisfies local designation and certificate-hash policy.

Proof of Security:

An attacker cannot substitute a topology node without forging the ADC signature. Query binding prevents a valid response for one issuer or serial query from being replayed as the answer to another query.

6.22 Topological Status Request and Response

Overview:

The topological status functions perform a signed liveness and status check between devices using topology node identities and certificates. The requester sends a signed status challenge for a remote node, and the responder returns a signed status result bound to the challenge and freshness state.

API:

aern_network_topological_status_request()
aern_network_topological_status_response()

Applies to:
ADC
APS
Client

Mathematical Description:

Let TL be the local topology node, TR be the remote topology node, and let st be the sequence and valid-time timestamp.

HSQσ = SignreqKpri(H(TL || TR || st))

Request(TL, TR) = (TL || TR || HSQσ)

The responder validates the requester certificate and node identity, then returns an authenticated status response:

HSRσ = SignrespKpri(H(TR || status || st))

Response(status) = (status || HSRσ)

The requester accepts the status result only if the responder signature, certificate chain, node serial, issuer, designation, sequence number, and timestamp validate.

Proof of Security:

The signed request prevents unauthenticated status probes from being confused with authorized topology maintenance. The signed response prevents an attacker from falsely asserting that a node is alive, revoked, or unavailable.

7. Security Analysis

AERN is designed to provide authenticated relay transport, confidentiality, integrity, replay resistance, and practical traffic-analysis mitigation in a controlled proxy domain. Relay sessions, route maps, payload headers, fragmentation state, return flags, dummy traffic, and backend transport boundaries are modeled explicitly as normative protocol mechanisms.

7.1 Introduction to Security Context

The Authenticated Encrypted Relay Network is intended for private or institutionally administered relay domains. The security posture differs from public volunteer anonymity networks because all participating proxy nodes are enrolled through a root-signed certificate model and synchronized through an ADC-managed topology. The principal security target is not open admission resistance, but authenticated participation, route confidentiality on the wire, strict replay protection, and reduced traffic observability inside a controlled infrastructure.

7.2 Comparative Analysis with TOR

TOR uses public volunteer relays and fixed circuits that are periodically rotated. AERN uses authenticated APS nodes and pre-established encrypted tunnels among synchronized proxy peers. The authenticated topology reduces Sybil and malicious-node exposure in the intended deployment model. The cost of this improvement is centralized trust in the ARS and ADC administrative model.

AERN also differs from TOR in the data phase. TOR hides route information through onion layers along a circuit. AERN uses a fully meshed APS tunnel substrate and carries the current route map inside an authenticated encrypted relay plaintext. The interior route may change per packet, while ingress and egress session endpoints remain stable for the logical session. This design supports efficient return traffic and reassembly while preventing a long-lived fixed interior route from defining the entire session path.

7.3 Security Strengths in Private Implementation

The closed-domain model provides several implementation-level strengths. First, an APS cannot participate in the topology without a certificate chaining to the ARS and registration with the ADC. Second, APS-to-APS traffic uses authenticated tunnel encryption and strict sequencing. Third, relay metadata in the header is encrypted inside the relay plaintext. Fourth, backend forwarding is separated from the relay cryptographic core, reducing protocol ambiguity around exit behavior.

AERN also defines traffic-shaping controls. Dummy relay packets and randomized ingress delay reduce idle-period observability and deterministic packet-injection timing. These controls are useful in practical sparse-flow environments but are not sufficient by themselves to provide mixnet-grade anonymity against a fully global adversary with precise timing visibility.

7.4 Node Authentication and Topological Integrity

Node authentication is enforced by root-signed certificates, ADC topology distribution, certificate hash binding, and signed control messages. The topology list binds each node address, issuer, serial number, expiration time, designation, and certificate hash. A route hint is meaningful only relative to this synchronized topology order. If a node is removed or revoked, its hint becomes invalid under the refreshed topology and peer tunnel state must be destroyed.

7.5 Channel Confidentiality and Integrity

Each APS tunnel encrypts and authenticates the relay plaintext between adjacent nodes. The authenticated context includes the outer header fields required for tunnel validation. The encrypted relay plaintext contains the length prefix, route map, relay payload header, and relay body. This construction provides confidentiality of route hints, session identifiers, fragment metadata, payload type, return flags, and backend payload contents against observers that do not control the decrypting APS endpoint.

7.6 Replay Resistance

Replay resistance is implemented by a combination of timestamp freshness and strict sequence validation. The current replay-window size is zero, and strict sequencing is enabled. A receiver accepts only the next expected tunnel sequence number. Duplicate, stale, reordered, or future-sequence packets are rejected. Fragment reordering is permitted only inside the terminal fragment cache after the enclosing tunnel packet has been authenticated.

7.7 Relay Session Security

The relay session-open procedure prevents backend delivery before the egress APS has explicitly accepted the session. The ingress maintains a pending state and queue until it receives a valid session-open acknowledgement. This prevents a first data packet from being delivered through an unconfirmed egress path. The return flag provides direction separation so that packets originating from the backend return path are not confused with outbound client packets.

7.8 Fragmentation Security

Fragmentation state is encrypted and session-bound. The fragment cache validates the session identifier, packet identifier, fragment sequence, fragment count, declared message length, payload type, reserved relay byte, and direction flag before reassembly. Incomplete or inconsistent fragment

sets expire or are discarded. Because fragmentation metadata is not cleartext, external observers cannot identify multi-fragment messages by header inspection alone.

7.9 Dummy Traffic and Timing Mitigation

Dummy packets provide cover traffic during periods of low utilization. They use ordinary relay framing and tunnel encryption and are discarded at the terminal APS. The traffic policy bounds dummy-packet emission by a local accounting window and suppresses dummy generation above the configured utilization ceiling. Random ingress delay adds small local timing uncertainty before packets enter the APS mesh.

These mitigations reduce simple traffic-volume and idle-period correlation, but they do not eliminate timing correlation by an adversary with complete observation of all client ingress links, all APS links, and all egress links. Stronger global-anonymity resistance would require additional batching, larger cover-traffic budgets, or mixnet-like scheduling that is outside the current low-latency AERN design.

7.10 Backend Transport Boundary

The backend transport callback boundary is a security boundary. AERN authenticates and protects relay transport up to serialized packet delivery at the ingress or egress boundary. The external destination, operating-system network device, userspace TCP or UDP stack, or application backend is not part of the AERN cryptographic relay proof. Implementations SHALL avoid logging or exporting relay session metadata at this boundary.

7.11 Potential Limitations and Mitigations

The full-mesh APS tunnel model has O(n^2) state growth. This is acceptable for moderate controlled domains and provides fast symmetric transport, but very large domains may require sharding, on-demand tunnel creation, or topology partitions. The compact route-hint encoding also limits a single active route-addressable APS set to 255 one-byte hints, so deployments with larger authenticated topologies SHALL partition the topology into route domains or define an extended route-hint profile before exceeding that active route-set size. RCS review maturity is also a consideration; implementations may define an approved symmetric fallback profile if external validation requires it.

The full-mesh APS tunnel model has O(n^2) state growth. This is acceptable for moderate controlled domains and provides fast symmetric transport, but very large domains may require sharding, on-demand tunnel creation, or topology partitions. RCS review maturity is also a consideration; implementations may define an approved symmetric fallback profile if external validation requires it.

7.12 Conclusion of Security Analysis

AERN provides a strong private-domain relay security model when the ARS, ADC, and APS nodes are administered under disciplined operational controls. The protocol defines explicit relay sessions, separates backend transport from the cryptographic core, enforces strict tunnel sequencing, and provides bounded dummy traffic and ingress-delay controls. These mechanisms strengthen the protocol model and reduce ambiguity in security analysis.

8. Cryptanalysis of the Authenticated Encrypted Relay Network

8.1 Adversary Model and Security Targets

The adversary model considers a global passive network observer, a local active attacker, and a node-compromise adversary. The network observer may see packet timing and fixed packet sizes on links it can monitor. The local active attacker may drop, delay, reorder, inject, or replay packets. The node-compromise adversary may compromise a subset of APS nodes and may later obtain long-term signing keys. A future quantum adversary is considered after protocol execution.

The required security goals are node authentication, channel confidentiality, channel integrity, replay resistance, route-map confidentiality on the wire, relay session binding, direction separation for return traffic, and practical resistance to traffic-volume analysis in sparse flows. Forward secrecy depends on the erasure of ephemeral KEM secrets and on the tunnel reinitialization policy used by the implementation.

8.2 Cryptographic Core

AERN uses post-quantum asymmetric primitives for certificate authentication and master encryption key establishment. The symmetric tunnel layer uses RCS with authenticated encryption semantics and SHAKE-based derivation. Certificates bind identity, topology designation, configuration, version, expiration time, issuer, serial number, and signature verification key. Control messages bind payloads to timestamps and sequence values by digital signature.

The session key derivation may be represented abstractly as k = KDF(ss || ctx), where ss is the KEM shared secret and ctx includes configuration, issuer, serial number, and role-separation context. Directional tunnel keys and nonces are derived independently for transmit and receive states. The tunnel encryption function is assumed to provide confidentiality and ciphertext integrity under unique synchronized state.

8.3 Handshake-Level Analysis

The MEK exchange authenticates the certificate chain and the signed key-exchange messages. A man-in-the-middle attacker must either forge the certificate chain, forge the device signature over the key-exchange transcript, or break the KEM. Under EUF-CMA security of the signature scheme and IND-CCA security of the KEM, the attacker cannot replace the public encapsulation key or ciphertext without detection.

The ADC topology messages are signed and certificate-bound. A malicious node cannot introduce a new topology entry unless it obtains a valid root-signed certificate and is accepted by the ADC. A stale topology message is rejected by freshness checks and sequence policy. A revoked node loses valid topology participation after the revocation update is processed.

8.4 Data-Phase Cryptanalysis

The wire-visible relay packet is fixed at 1500 bytes and contains a 22-byte outer tunnel header and an opaque ciphertext region. The route path, relay payload header, fragment sequence, fragment count, session identifier, packet identifier, payload type, reserved relay byte, and return flag are inside the encrypted relay plaintext.

Let P = L || RM || RH || B || pad, where RM is the sixteen-byte consumed route path and RH is the thirty-two-byte relay payload header. The tunnel ciphertext is C = AEAD_K(Hdr, P). Under the assumption that the authenticated tunnel cipher is IND-CPA secure and INT-CTXT secure for synchronized nonces and keys, C hides P from any observer not holding the relevant tunnel receive state. Under integrity, an adversary cannot alter RM or RH to redirect, reclassify, fragment, or reverse a packet without causing authentication failure at a subsequent processing step.

8.5 Replay and Freshness Analysis

The implementation uses strict sequencing rather than a sliding replay window. Let seq_rx be the next expected receive sequence. Acceptance requires seq = seq_rx and a valid packet timestamp. A replayed packet has a sequence below seq_rx and is rejected. A reordered future packet has a sequence above seq_rx and is rejected at the tunnel layer. This is appropriate for ordered APS tunnel transport.

Fragment reordering does not contradict strict tunnel sequencing because fragments are reorderable only after successful tunnel authentication and only inside the relay fragment cache. The fragment cache is keyed by session and packet identifiers and enforces timeout and consistency checks. The security boundary is therefore layered: strict order at the tunnel level, controlled reassembly at the encrypted payload level.

8.6 Anonymity and Traffic-Analysis Resistance

AERN provides practical anonymity by hiding the external client address from the destination, encrypting route and session metadata on the wire, using fixed-size packets, changing interior routes per packet, and optionally adding dummy traffic and ingress delay. A passive external observer sees fixed MTU-sized encrypted packets rather than variable clear payloads or clear routing headers.

The ingress and egress APS nodes remain stable for the logical relay session, because that stability is required for session-open acknowledgement, backend delivery, return traffic, and fragment reassembly. Interior hops are randomized per packet. If both ingress and egress are compromised, traffic linkage becomes easier. If at least one endpoint remains honest and the adversary lacks complete timing visibility, per-packet interior rerouting and fixed packet sizing raise the cost of correlation.

8.7 Dummy Traffic Limits

Dummy traffic reduces sparse-flow and idle-period observability, but it is not a universal anonymity mechanism. A low-latency system that emits bounded dummy traffic cannot provide the same protection as a high-latency mixnet with global batching and cover traffic. The correct cryptanalytic claim is that dummy traffic improves resistance to simple traffic-volume analysis and makes idle tunnel periods less distinguishable, not that it defeats a global timing adversary.

8.8 Comparison with Established Anonymous Overlays

Compared with TOR, AERN trades open public participation for a controlled authenticated domain. Compared with mixnets, AERN favors low latency and backend interoperability over strong batching anonymity. Compared with ordinary VPNs, AERN adds multi-hop authenticated relay transport, encrypted route metadata, topology-controlled proxy membership, and per-packet interior route variation. Its primary trade-offs are centralized trust and full-mesh state growth.

8.9 Potential Weaknesses and Mitigations

ARS compromise can subvert future enrollment and certificate issuance. Mitigations include root isolation, threshold root signing, short-lived ADC certificates, auditable signing operations, and strict revocation processing. ADC compromise can corrupt topology distribution; mitigations include signed topology epochs, independent topology audit, and proxy-side convergence verification.

Compromised APS nodes can observe relay plaintext at their hop after decryption and can see the route map for packets they process. They cannot read packets carried over tunnels for which they are not an endpoint. Compromise of both ingress and egress enables stronger correlation. Mitigations include administrative separation of proxies, multi-domain chaining, higher minimum hop counts where latency permits, and stronger dummy or batching policies.

Implementation risks include side-channel behavior in asymmetric primitives, insufficient entropy, inconsistent clock synchronization, and backend logging. Implementations SHALL use constant-time cryptographic implementations where available, validated random sources, reliable time synchronization, strict input validation, and backend logging policies that do not retain relay-flow metadata.

8.10 Conclusion of Cryptanalysis

Under the stated assumptions, AERN provides authenticated node participation, confidential and integrity-protected APS tunnel transport, strict replay rejection, encrypted route and fragment metadata, explicit relay session binding, and bounded traffic-shaping support. The remaining limitations are operational trust centralization, endpoint-compromise correlation, and the inherent limits of low-latency cover traffic.

9. Conclusion

AERN defines an authenticated, encrypted relay network for controlled proxy domains. Its architecture combines a root-signed trust model, ADC-managed topology, post-quantum key establishment and signatures, pre-established symmetric APS tunnels, fixed-size relay packets, encrypted route metadata, explicit ingress-to-egress sessions, and backend-neutral transport integration.

The principal strengths of the current revision are the authenticated private infrastructure, the removal of clear route and fragment metadata from the relay header, the strict tunnel replay policy, the explicit relay session-open state machine, and the addition of dummy traffic and ingress-delay controls. These refinements make the specification more accurately aligned with the implementation and reduce ambiguity in the protocol security model.

AERN is not a public volunteer anonymity system and should not be evaluated as one. It is a controlled, certificate-enforced relay domain intended for environments where proxy membership, revocation, topology synchronization, and cryptographic implementation quality can be administered. In that deployment model, AERN offers stronger node admission control, post-quantum posture, and relay metadata protection than conventional VPN systems and many classical anonymizing overlays.

The important limitations are also explicit. The ARS and ADC are high-value trust components. The full-mesh tunnel model grows with the number of APS nodes. Dummy traffic and short randomized ingress delay reduce practical observability but do not provide mixnet-grade resistance against a fully global timing adversary. Backend transport adapters must be implemented carefully to avoid logging or leaking relay metadata outside the cryptographic core.

The AERN specification should therefore be read as a protocol for authenticated, post-quantum-ready, low-latency relay domains with strong encrypted-tunnel semantics and practical traffic-analysis mitigation. Continued work should focus on validation of the symmetric cipher profile, public test vectors, formalized topology epochs, backend conformance tests, and operational guidance for ARS and ADC hardening.

Appendix A. AERN Conformance Profile

This appendix defines the AERN conformance profile. It is normative for deployments that implement the compact single-domain relay profile described in this specification. It specifies the protocol architecture, packet and state boundaries, and validation obligations required for interoperable AERN implementations.

A.1 Source module responsibility map

The implementation is organized as a set of role modules, cryptographic state modules, topology modules, relay modules, and administrative support modules. Conforming implementations MAY use a different file layout, but SHALL preserve the state boundaries and failure semantics described in this specification.

Module	Primary responsibility	Conformance requirement
aern.h	Global constants, protocol sizes, enumerations, certificate structures, connection state, and packet state.	Protocol-size constants in this file define the active wire and serialized sizes unless explicitly overridden by a future profile.
certificate.c/h	Root and child certificate generation, serialization, decoding, validation, comparison, and root signing.	Certificate acceptance requires structural validity, accepted algorithm and version fields, nonzero required fields, expiration validity, and valid root signature where applicable.
topology.c/h	Topology node encoding, topology-list storage, canonical sorting, topology hashing, version management, lookup, insertion, update, and removal.	Topology mutation SHALL increment the monotonic version. Incoming topology versions SHALL be rejected when stale under the local version policy.
network.c/h	Administrative message families, registration, incremental update, convergence, fragment-key request/response, MFK exchange, remote signing, resignation, revocation, topology query, and topology status.	Administrative messages SHALL bind sender role, certificate validity, sequence or message freshness where implemented, timestamp validity, signed payload content, and topology version where the message carries a version.
mek.c/h	Mesh encryption key request/response handling, peer tunnel cipher table, synchronization state, rekey thresholds, and APS mesh peer state.	Each mesh peer SHALL have independent transmit and receive cipher state and SHALL be marked synchronized only after authenticated key establishment and local state installation complete.

route.c/h	Relay packet serialization, encrypted consumed-path route maps, relay payload headers, session-open and acknowledgement payloads, fixed-size relay packet processing, dummy traffic, ingress delay, and backend callback integration.	Relay metadata SHALL remain inside the encrypted relay plaintext. Backend delivery SHALL occur only after tunnel authentication, route/session validation, and reassembly when required.
relaysession.c/h	Relay session cache lifecycle and session state transitions.	A non-dummy data packet SHALL NOT be released from pending state until a valid session-open acknowledgement is accepted for the same session identifier and egress context.
relayqueue.c/h	Ingress pending queue and delayed packet queue management.	Queued packets SHALL expire according to local timeout policy and SHALL be erased on session failure, tunnel failure, or teardown.
fragment.c/h	Fragment table and fragment cache functions for encrypted relay payload reassembly.	Fragment metadata SHALL be accepted only after tunnel authentication and SHALL be bound to session identifier, packet identifier, direction, payload type, flags, reserved byte, count, and reconstructed length.
adc.c/h, aps.c/h, ars.c/h, client.c/h	Application role orchestration for the domain controller, proxy server, root security server, and client.	Role modules SHALL enforce the allowed message families and designation checks for their role.
admin.c/h, menu.c/h, logger.c/h, server.c/h, lifecycle.c/h	Console, logging, server lifecycle, administrative cleanup, resignation, revocation, keepalive, and error-context support.	Operational functions SHALL NOT log route paths, session identifiers, fragment identifiers, packet identifiers, or backend payload bytes in ordinary logs.

A.2 Active constants and derived relay capacities

The following constants define the active compact relay profile. They are included here so that independent implementations can verify the same packet geometry without inferring it from prose.

Symbol	Value	Meaning
AERN_NETWORK_CONNECTION_MTU	1500 bytes	Fixed relay wire packet size.

AERN_PACKET_HEADER_SIZE	22 bytes	Visible per-hop tunnel header authenticated as associated data.
AERN_RELAY_CIPHERTEXT_SIZE	1478 bytes	Encrypted tunnel ciphertext region after the outer header.
AERN_CRYPTO_SYMMETRIC_MAC_SIZE	32 bytes	Authentication material included in the ciphertext-region accounting.
AERN_RELAY_PLAINTEXT_SIZE	1446 bytes	Decrypted relay plaintext capacity.
AERN_LEN_PREFIX_SIZE	2 bytes	Encrypted used-length prefix.
AERN_ROUTE_PATH_SIZE	16 bytes	Encrypted consumed-next-hop route path.
AERN_MAX_USER_PAYLOAD	1428 bytes	Relay content region after length prefix and route path.
AERN_RELAY_PAYLOAD_HEADER_SIZE	32 bytes	Encrypted relay session, packet, fragment, payload-type, reserved, and flags header.
AERN_RELAY_DATA_PAYLOAD_SIZE / AERN_FRAG_CHUNK_SIZE	1396 bytes	Maximum backend payload carried by one relay fragment.
AERN_RELAY_SESSION_OPEN_SIZE	36 bytes	Encrypted session-open payload.
AERN_RELAY_SESSION_OPEN_ACK_SIZE	12 bytes	Encrypted session-open acknowledgement payload.
AERN_ROUTE_PATH_SIZE route hints	16 one-byte hints	One origin hint and up to fifteen future-hop slots.
Compact route-addressable APS set	255 APS entries	One-byte one-based hints reserve zero as the consumed and unused value.

A.3 Topology versioning and stale-update rejection

The current topology state contains a uint64_t version field directly in aern_topology_list_state. Version zero denotes an uninitialized or disposed topology list. A topology mutation caused by insertion, replacement, removal, revocation, register-update acceptance, or lifecycle cleanup SHALL increment the version, with wrap to zero avoided by restoring the value to one. A received topology version is accepted only when it is strictly greater than the local version for update paths that require monotonic advancement. A stale version SHALL be rejected and SHALL NOT replace the local topology list, peer cache, or route-hint resolution state.

The versioned register-update response is the active response profile for ADC transfer of topology state with monotonic freshness. The ADC serializes the topology version with the topology list and signs the response under its child certificate. A client or APS receiving the response SHALL validate the ADC certificate, response signature, timestamp, structural topology list, and topology version before replacing local topology state.

A.4 Mesh encryption key state and peer synchronization

The mesh encryption key module defines a peer-synchronization state machine with the states none, synchronizing, synchronized, failed, and expired. A peer SHALL be treated as route-usable only after synchronization reaches the synchronized state and the connection state contains initialized transmit and receive RCS cipher instances. Synchronization failure, certificate expiration, topology revocation, repeated authentication failure, or key expiration SHALL remove the peer from route eligibility until authenticated renewal completes.

Rekey thresholds are represented as a soft threshold, grace threshold, and packet threshold. These thresholds are local policy controls for initiating renewal before an existing tunnel becomes unsafe or administratively stale. A conforming implementation SHALL NOT reuse a peer tunnel beyond local expiration policy, SHALL erase superseded cipher state after renewal, and SHALL reject packets that arrive under a peer state marked failed or expired.

A.5 Relay queues, delayed ingress, and teardown hygiene

The relay queue subsystem separates pending-session buffering from randomized ingress delay. Pending queue entries exist because non-dummy data cannot be delivered until the egress has accepted a session-open message and returned a valid acknowledgement. Delayed ingress entries exist only to reduce deterministic timing linkage before mesh injection. Both queues are local relay state. They SHALL be bounded, SHALL expire stale entries, and SHALL be cleared when a session fails, an acknowledgement is rejected, a tunnel is torn down, or administrative revocation removes a route peer.

The relay session cache distinguishes none, pending, active, closing, expired, and failed states. A session-close payload, backend failure, pending timeout, idle timeout, tunnel failure, or topology revocation SHALL transition the affected cache entry to a teardown path and SHALL erase queue references and fragment-cache references associated with the session.

A.6 Fragment-cache conformance

Fragmentation is strictly an encrypted relay-layer function. A fragment table entry SHALL NOT be created from unauthenticated ciphertext or from a packet with an invalid timestamp, unexpected tunnel sequence, invalid payload header, nonzero reserved relay byte, invalid fragment count, invalid fragment sequence, inconsistent flags, or inconsistent direction. A complete fragment set SHALL be delivered exactly once, and incomplete sets SHALL expire according to AERN_RELAY_FRAGMENT_CACHE_TIMEOUT_MILLISECONDS. The current maximum fragment count is 4096. Deployments that raise this limit SHALL account for memory pressure and denial-of-service exposure.

A.7 Backend callback boundary

The active implementation uses backend callbacks rather than direct exit-forwarding as the protocol boundary. The cryptographic relay core terminates at delivery of an authenticated, session-valid, and

reassembled serialized packet to the configured egress or ingress callback. A backend MAY implement a TUN or TAP device, raw socket path, userspace TCP or UDP stack, packet filter, or application-specific adapter. A backend SHALL NOT receive route paths, route cursor state, tunnel sequence numbers, session-cache internals, fragment-cache internals, or dummy payloads.

A.8 Conformance tests required for this revision

- Certificate tests SHALL cover root and child serialization, text encoding, root-signature verification, expiration rejection, algorithm and version rejection, designation rejection, and malformed input rejection.
- Topology tests SHALL cover serial-number ordering, topology hashing, version increment, stale-version rejection, add, update, remove, revoke, ADC lookup, APS lookup, and client topology replacement.
- Network-message tests SHALL cover every administrative message family, valid paths, truncated messages, corrupted signatures, stale timestamps, invalid sender designation, malformed topology payloads, and no unintended state mutation after rejection.
- MEK and tunnel tests SHALL cover authenticated key exchange, peer synchronization state transitions, transmit/receive direction separation, sequence initialization, authentication failure accounting, rekey thresholds, expiration, and peer teardown.
- Relay tests SHALL cover route generation, one-byte hint bounds, nonconsecutive repetition policy, next-hop consumption, terminal detection, fixed packet sizing, encrypted relay header parsing, session-open, session-open acknowledgement, pending queue release, session close, backend callback isolation, return-flag handling, dummy discard, ingress delay expiry, fragmentation, reassembly, and replay rejection.
- Virtual-network tests SHALL instantiate ARS, ADC, at least four APS nodes, a client, and a backend callback path. The test SHALL verify control-plane initialization, APS mesh synchronization, client registration, session-open, opaque outbound delivery, return delivery, dummy traffic handling, and teardown under fault injection.