Authenticated Encrypted Relay Network - AERN

Revision 1.0a, March 28, 2025

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 **Page**

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 john.underhill@protonmail.com
AERN, the algorithm constituting the AERN anonymous network system is patent pending, and is owned by John G. Underhill and the QRCS Corporation.

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.
Exit nodes can either perform a client-to-server function; in which the message packet is decrypted and sent to a server that is not part of the network (ex. HTTP server), or client-to-client function in which an underlying encrypted tunnel between remote clients is established which operates underneath the proxy circuit.
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.

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.

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.

1.1 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 (Anonymous 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. Packet headers are encrypted along with the payload between devices, a receiving device derives the correct connection-set symmetric cipher instances via the IP address of the sender.
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

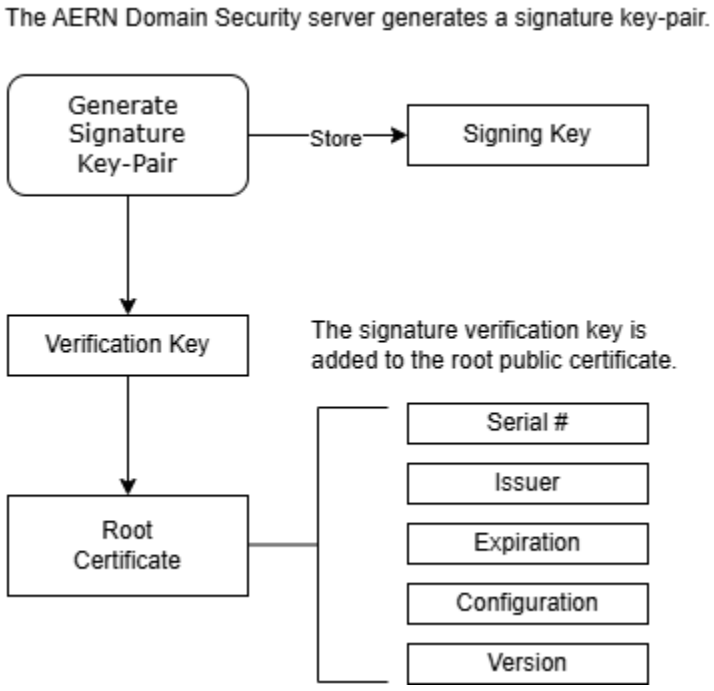


Figure 5.4.1 Root certificate generation.

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
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.
configuration	4 bytes	Uint32	The algorithm configuration identifier.
version	4 bytes	Uint32	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

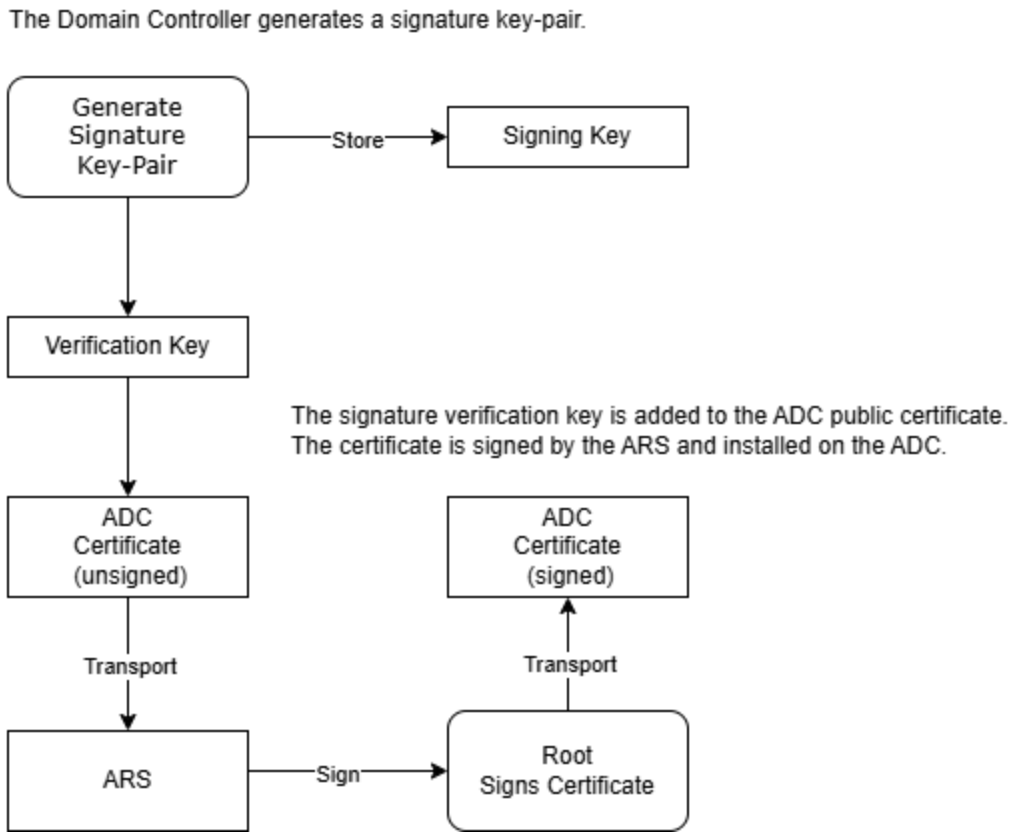


Figure 5.4.2a ADC certificate initialization

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's root signed hash.
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's serial number.
expiration	16 bytes	Uint8 Array	The from and to certificate expiration times.
designation	4 bytes	enum	The certificate type designation.
configuration	4 bytes	Uint32	The algorithm configuration identifier.
version	4 bytes	Uint32	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.

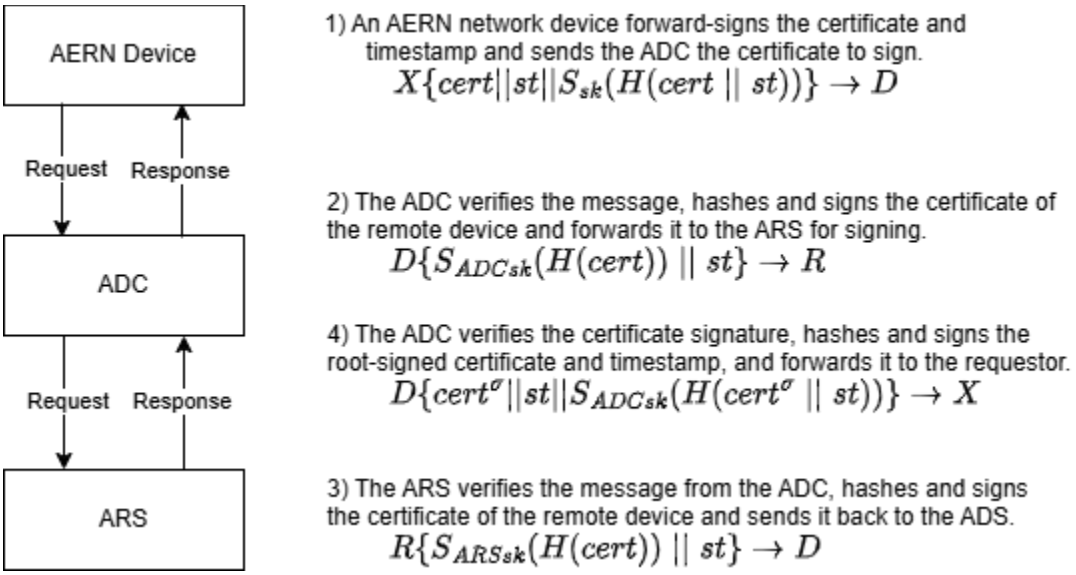


Figure 5.4.2b ADC proxy signing

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 structs with pointers to the connection structures (which contain the send and receive channels symmetric cipher states) and socket structures (containing the socket pointer for each connected device) are also arranged in this identical sequence, which corresponds to the index number contained in a ***route hint***, an element in the ***route map*** field of the AERN packet header.

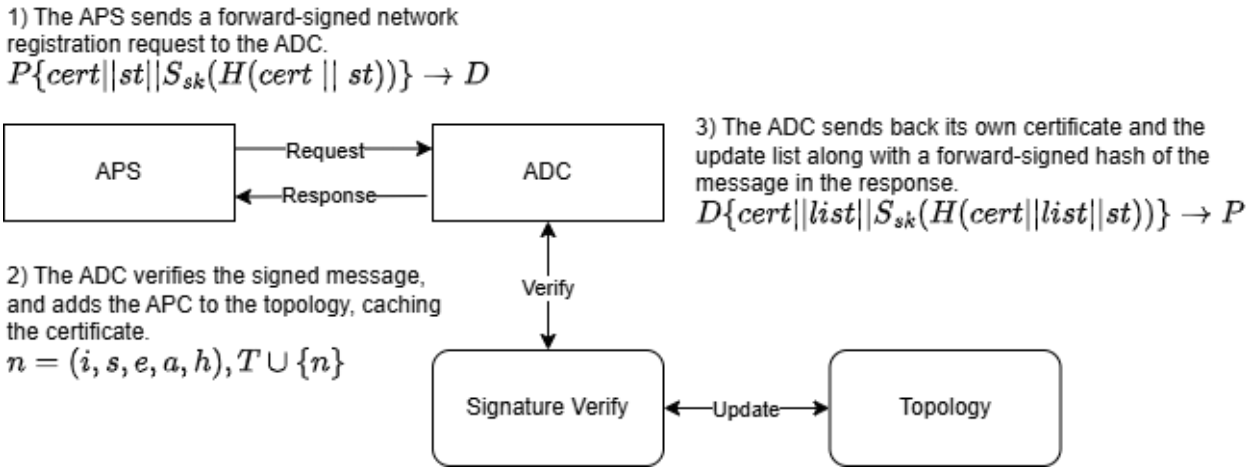


Figure 5.4.4a APS network registration.

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.
mapkey	32 bytes	Uint8 Array	The node's map key.
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	4 bytes	enum	The certificate type designation.

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		Description
topology	8 bits	Uint8 Array	The serialized topology list.
count	4 bytes	Uint32	The list node count.
lhash	32 bytes	Uint8 Array	A hash of the topology list.

Table 5.4.4b: The aern_topological_list structure.

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.

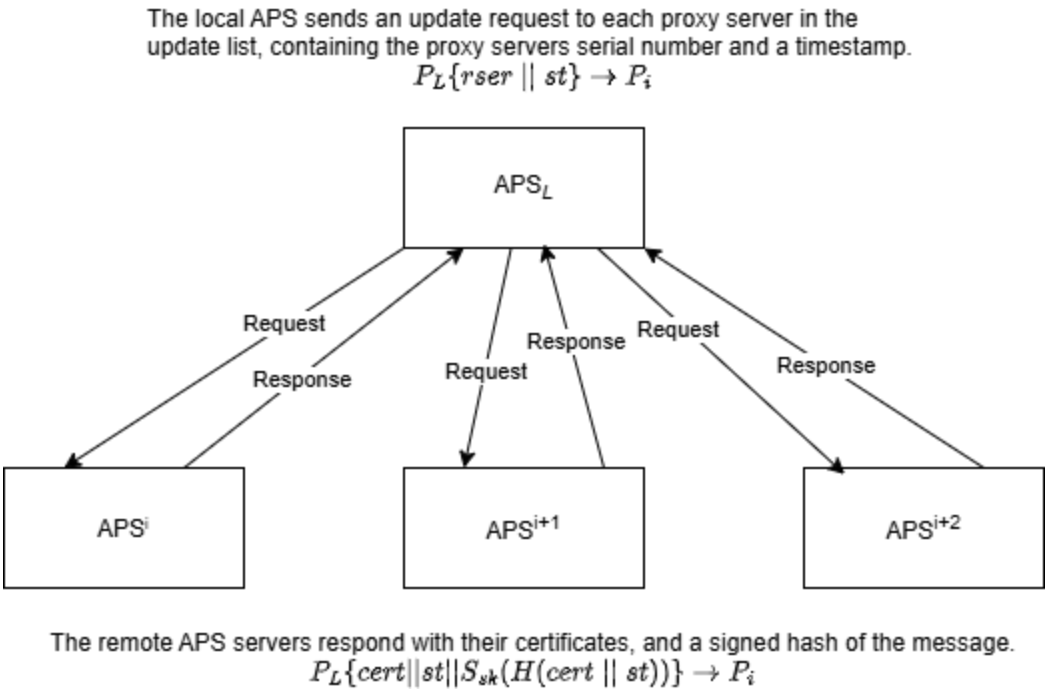


Figure 5.4.4b APS proxy update.

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.

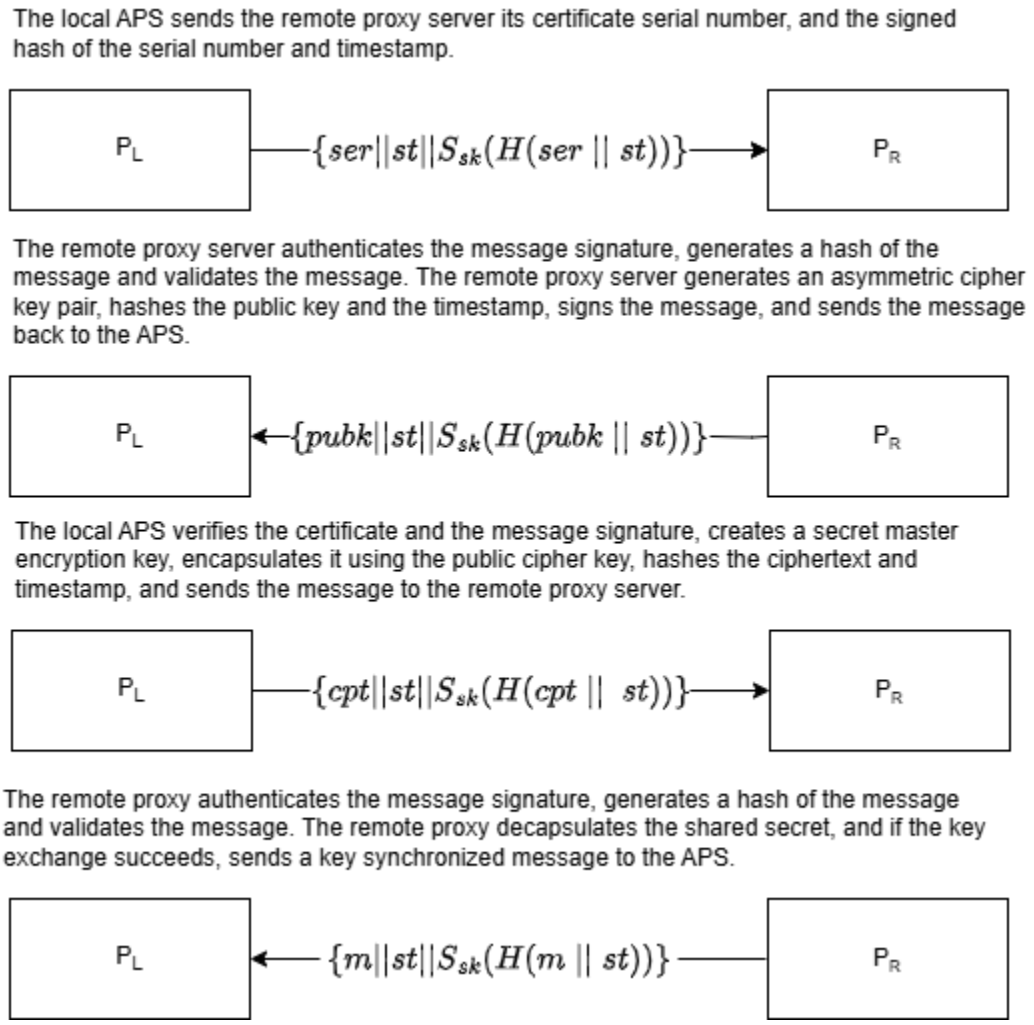


Figure 5.4.4c APS to proxy master key exchange.

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 (+/- 30 seconds). 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:

Name	Size	Type	Description
target	104 bytes	struct	The remote nodes connected socket instance.
rxcpr	1184 bytes	struct	The receive channel cipher state.
txcpr	1184 bytes	struct	The transmit channel cipher state.
rxseq	8 bytes	Uint64	The receive channel packet sequence number.
txseq	8 bytes	Uint64	The transmit channel packet sequence number.
instance	4 bytes	Uint32	The structure instance number.
exflag	4 bytes	enum	The connection state flag.

Table 5.4.4c: The aern_connection state.

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.

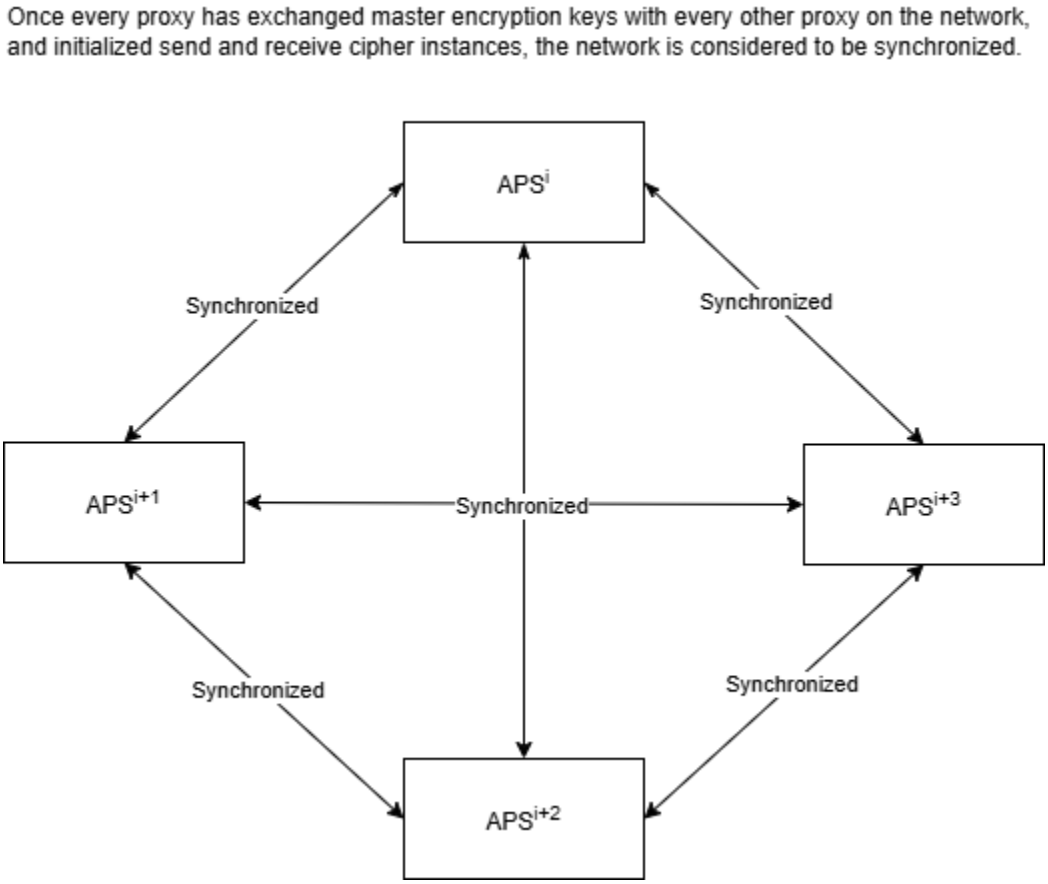


Figure 5.4.4e synchronized full-mesh proxy network.

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 ADS 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.

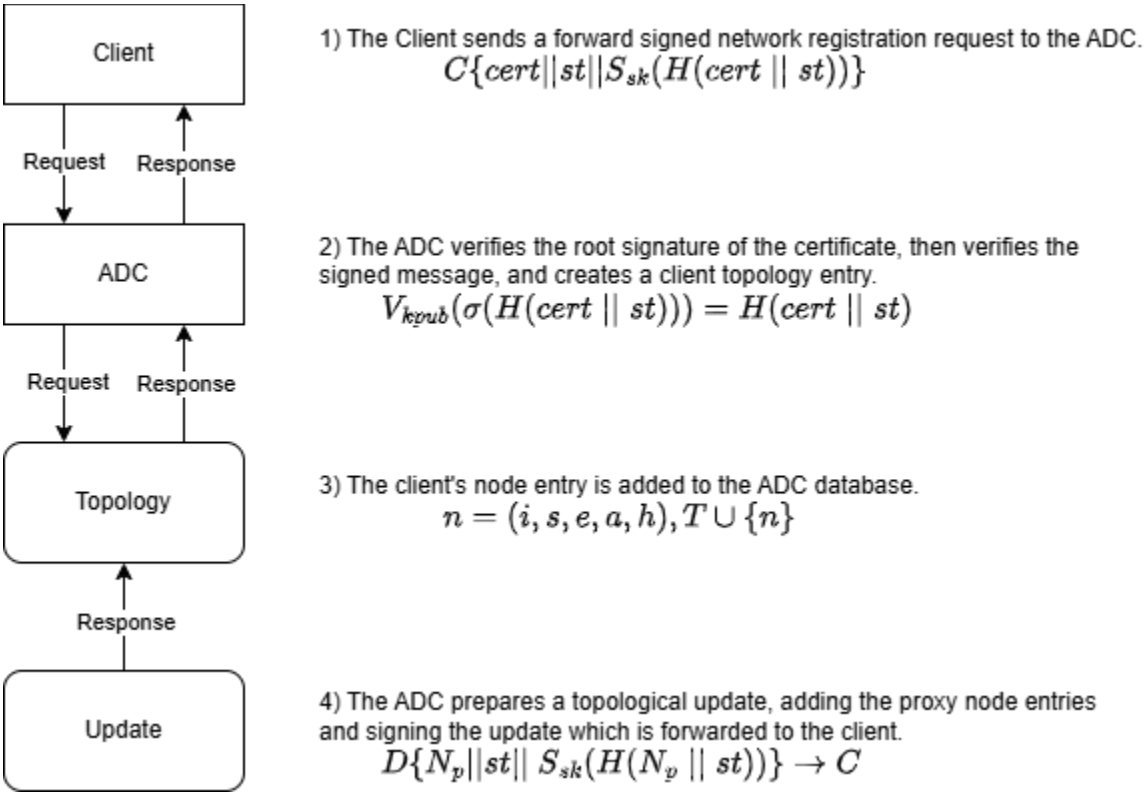


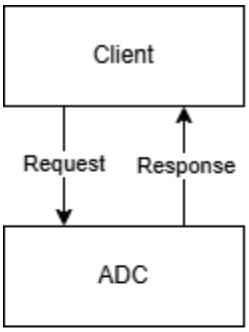
Figure 5.4.5a client domain registration request.

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.

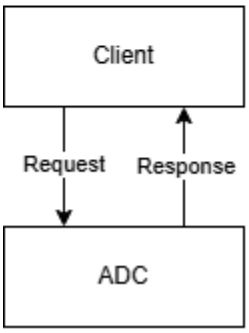


1) The Client sends a forward signed network join request to the ADC.

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

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.

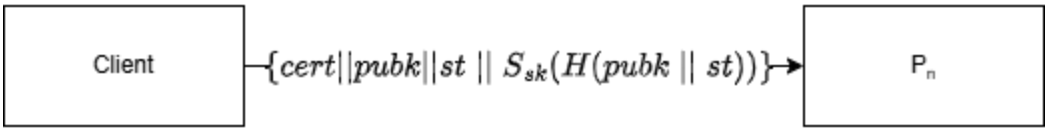
$$D\{N_p||st||S_{sk}(H(N_p \mid\mid st))\} \rightarrow C$$

Figure 5.4.5b client domain join request.

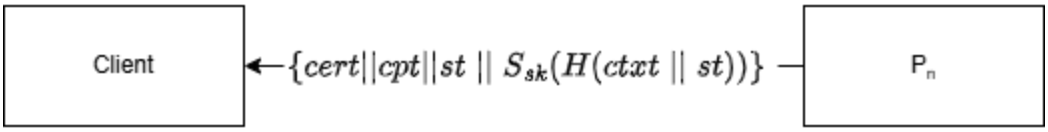
5.4.6 Client Entry Node Selection and Connection

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.

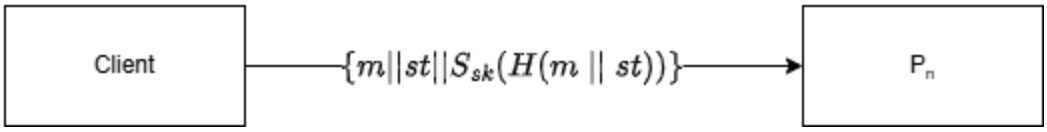


Figure 5.4.6a client to proxy key exchange.

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 sends a packet to be transported through the proxy network to the entry node, the entry node generates a *route map*. This map is a set of proxy nodes that comprise the network circuit between the entry node and the exit node. The length of this path is a ranged random number between three, the ***minimum hops*** number of nodes including the entry and exit nodes, and ***maximum hops***, a tunable maximum hop value that defaults at sixteen hops. This first route map generation determines the exit node for the duration of the session, the entry and exit nodes remains unchanged until the session is torn down, but the intermediate nodes are changed each time a packet enters the proxy network.

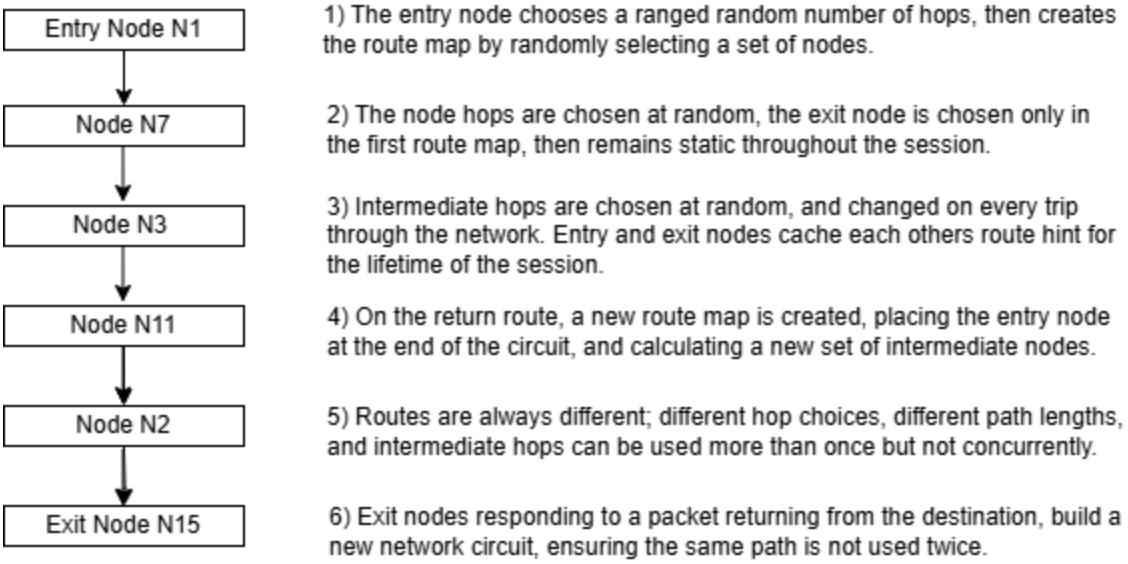


Figure 5.4.7a route map generation.

The route maps are created by the entry node, and when traversing back through the network the exit node. Any proxy can be chosen as an intermediate node, even more than once (though not consecutively). The maximum number of hops is set at a fixed value, as is the minimum number by tunable constants which default at sixteen and three. The minimum hops include the entry node and exit node, so a path must minimally be at least three hops by default. The entry node selects each intermediate node in the path using a ranged random number generator; $\boldsymbol{n = R(min, max)}$, to a maximum of the random number $\mathbf{H_n}$, the number of hops in the route.

Each hop in the route map is a ***route hint***, a 16-bit integer representing the natural index number in the proxy ***network node*** list. Every proxy in the network has a copy of an identical topology list administrated over by the domain controller, a parallel list comprised of network node structures is created that matches the topology ordering.

A given node is selected from this list using this route hint. When a device receives a routing instruction from the map, it looks up the destination node in network node list at the index that corresponds to the route hint integer, and retrieves the network node structure. The network node structure contains both the ***socket state*** and the ***encryption connection*** state. The sending device loads the corresponding symmetric cipher instance and encrypts the packet and sends it to the next hop. Decryption happens at the receiving node, by looking up the socket instance number from the receiving socket structure, loading the corresponding receive channel symmetric cipher instance for that proxy-to-proxy tunnel, and decrypting the packet.

Note: route maps are encrypted along with the packet payload. When a proxy receives a message, it looks up the symmetric cipher instances that correspond to that IP host. In this way the route map is not needed to decrypt an incoming packet, but only to look up the next hop destination. Virtual tunnels operate within their own unique per-instance threads, with the

associated socket and network connection structures passed to the thread as the state parameter in a ***network node*** structure.

Socket Structure

The socket structure contains information about a connection with a remote device:

Name	Size	Type	Description
connection	8 bytes	socket_t	The socket instance pointer.
address	65 bytes	Uint8 Array	The IPv4 or IPv6 address array.
instance	4 bytes	Uint32	The socket instance number.
port	2 bytes	Uint16	The protocol port number.
address_family	enum	4 byte	The IP address family.
connection_status	enum	4 byte	The connection state.
socket_protocol	enum	4 byte	The socket protocol type.
socket_transport	enum	4 byte	The socket transport type.

Table 5.4.7a: The qsc_socket structure.

Network Node

The network node structure contains the socket structure and the network connection structure:

Name	Size	Type	Description
sock	Variable	socket structure	The socket structure.
conn	Variable	connection structure	The encryption connection structure.

Table 5.4.7b: The aern_network_node structure.

Route Map

The route map structure contains the route hints that are hops along a network path:

Name	Size	Type	Description
rmap	Variable	Uint8 Array	The route map, max hops times 2 bytes in size.
ridx	1 byte	Uint16	The route index, current index along the route.

Table 5.4.7c: The aern_route_map structure.

The entry node (or exit node on reverse travel) generates a route map, first it selects the number of hops in the path. The number of hops can range between AERN_ROUTE_MINIMUM_HOPS which is the entry node plus two hops; three hops in total, to a maximum of AERN_ROUTE_MAXIMUM_HOPS, which is 16 by default, but can be tuned to a higher number. We recommend that the maximum hop count not be lowered to less than 8 to maximize

timing obfuscation, and not more than is tolerable by the underlying applications message delay sensitivity.

The entry node then selects the nodes that will be the hops along the path randomly; a node can be used more than once but not concurrently.

Route maps consist of a set of node hints; a 16-bit integer that corresponds to the nodes index in the network node list common to every proxy server, and a route index; the current position in the map. The network node is fetched from the node list, the corresponding cipher instance encrypts the message including the packet header, increments the route index counter, and sends the message and header to the next hop node.

5.4.8 Network Traversal

Messages sent between the client and the server, or client to client, may exceed the packet size boundary of fifteen hundred bytes. The message is decrypted, buffered until the last fragment has been received, and then transmitted to the server. At the entry point, the client is responsible for disassembling messages along maximum length boundaries, and reassembling messages that have been fragmented by the exit node on an inverse packet flow. The exit node reassembles fragmented messages, caching the segments until the full packet is received, then sending the complete and decrypted message to the server. Likewise, the client caches segmented messages, and once the final segment is received, reassembles the fragments and passes the complete message up through the application stack. Packets containing a fragmented message are numbered using the fragmented sequence flag of the AERN packet header.

The AERN packet header is 54 bytes in length, and contains:

1. The **Packet Flag**, the type of message contained in the packet; this can be any one of the key-exchange stage flags, a message flag, or an error flag.
2. The **Packet Sequence**, this indicates the sequence number of the packet in the exchange.
3. The **Fragment Sequence,** zero if the packet is not fragmented, maximum length (~0) if the last packet in the sequence, or an integer representing the fragment sequence.
4. The **Message Size**, this is the size in bytes of the message payload.
5. The **UTC time**, the time the packet was created, used in an anti-replay attack mechanism.

The message is a variable sized array, up to QSMP_MESSAGE_MAX in size.

Packet Header

The AERN packet header, containing the message type flag, sequence number, fragment sequence, route map, and timestamp.

Packet Flag 1 byte	Packet Sequence 8 bytes	Fragment Sequence 4 bytes	Route Map 33 bytes	Time Stamp 8 bytes
Packet Payload 1446 bytes				

Table 5.4.8a: The AERN packet structure.

This packet structure is used for the inter-proxy relay communications stream.

Flag Types

The following is a list of packet flag types used by AERN:

Flag Name	Numerical Value	Flag Purpose
None	0x00	No flag was specified, the default value.
Encrypted Message	0x01	The message is part of a data stream.
Connection Terminated	0x02	The connection is to be terminated.
Keep Alive Request	0x03	The packet contains a keep alive request.
Keep Alive Response	0x04	The packet contains a keep alive response.
Error Condition	0xFF	The connection experienced an error.

Table 5.4.8b: Packet header flag types.

Error Types

The following is a list of error messages used by AERN:

Error Name	Numerical Value	Description
None	0x00	No error condition was detected.
Authentication Failure	0x01	The symmetric cipher had an authentication failure.
Bad Keep Alive	0x02	The keep alive check failed.
Channel Down	0x03	The communications channel has failed.
Connection Failure	0x04	The device could not make a connection to the remote host.
Connect Failure	0x05	The transmission failed at the KEX connection phase.
Decapsulation Failure	0x06	The asymmetric cipher failed to decapsulate the shared secret.
Establish Failure	0x07	The transmission failed at the KEX establish phase.

Exstart Failure	0x08	The transmission failed at the KEX exstart phase.
Exchange Failure	0x09	The transmission failed at the KEX exchange phase.
Hash Invalid	0x0A	The public-key hash is invalid.
Invalid Input	0x0B	The expected input was invalid.
Invalid Request	0x0C	The packet flag was unexpected.
Keep Alive Expired	0x0D	The keep alive has expired with no response.
Key Expired	0x0E	The QSMP public key has expired.
Key Unrecognized	0x0F	The key identity is unrecognized.
Packet Un-Sequenced	0x10	The packet was received out of sequence.
Random Failure	0x11	The random generator has failed.
Receive Failure	0x12	The receiver failed at the network layer.
Transmit Failure	0x13	The transmitter failed at the network layer.
Verify Failure	0x14	The expected data could not be verified.
Unknown Protocol	0x15	The protocol string was not recognized.
Listener Failure	0x16	The listener function failed to initialize.
Accept Failure	0x17	The socket accept function returned an error.
Hosts Exceeded	0x18	The server has run out of socket connections.
Allocation Failure	0x19	The server has run out of memory.
Decryption Failure	0x1A	The decryption authentication has failed.

Table 5.4.8c: Error type messages.

Messages are sent from the entry node to the next hop proxy server through the encrypted tunnel that they share. They are encrypted by the sender and decrypted by the receiver, then re-encrypted by the tunnel interface shared between the receiver and its next hop peer in the route map. A message travels through the circuit encrypted and decrypted by each hop until it traverses the entire circuit and reaches the exit node. No metadata is stored during this trip, and no information is logged on the proxy servers about traffic flows themselves. Logging SHALL be restricted to errors and failures occurring on the proxy server, and SHALL NOT include any information about individual traffic flows. Metadata SHALL NOT be stored on the proxy server, all activity from traffic flows is ephemeral, and is erased immediately after the data exits the proxy server, with the exceptions of the entry node retaining the exit node route hint to construct future route maps for a session, and that the exit node shall retain the entry node route hint to construct future route maps for inverse traffic flows, these route hints are retained for the lifetime

of a session, and erased after a disconnect message is sent by the client, or a session timeout timer is exceeded.

When a message reaches the exit node, if it requires reassembly, it is cached until all of the message fragments have been received and the full packet is assembled, then it is sent to the destination device. AERN treats the destination as having no knowledge of the proxy network and simply forwards the packet like a typical VPN server.

5.4.9 Disconnects, Errors, and Session Teardown

When a client disconnects from the network, typically by choosing a disconnection option or closing the application software, it sends a disconnect message to the entry node. The entry node forwards a ***client disconnect*** message through the proxy network to the exit node, in the same way it would forward any other message; with a random route-map and a payload of random bytes to fill a full 1500-byte message. Once the packet leaves the entry node, the entry node session cache is cleared, erasing the exit node address on the entry node (and entry node address on the exit node), and the encrypted tunnel between the client and the entry node is gracefully torn down and the state erased. If however, the exit is not graceful, for example the VPN application software is closed suddenly and the disconnect message is not sent, the entry node executes a session timeout shutdown after a timer running on the entry node has been exceeded. A timer is started when the client to entry node connection is first established. This timer is reset every time a packet is sent through the entry node for a session, and counts up to a maximum elapsed time; AERN_SESSION_TIMEOUT which is a tunable maximum elapsed time that triggers a client disconnect from the entry node if the timeout is exceeded. The default timeout period is 1800 seconds, or thirty minutes, after that amount of time if no traffic has transited the entry node either from the client or an inverse traffic flow from the server, then the timeout expires and the client disconnect message is sent, tearing down the connection and disconnecting the client from the network.

In the event that a proxy server can not connect to its next hop address, that the send function times out or that the device is not online, so long as that next hop is not the last device in the proxy chain; the entry or exit node, it can forward the packet to the next node in the sequence, bypassing the unresponsive proxy server. When a proxy server discovers an unresponsive peer, it forwards the error to the ADC. The ADC begins a ***node alive*** check, where every 10 seconds for a maximum of 300 seconds, it sends a keepalive request to the affected proxy node. If the proxy node responds within the timeout period, the *up* status is logged along with the failure, and network operation resumes. However, if the affected proxy server does not respond within the timeout threshold, the ADS sends a device ***revocation broadcast*** to every device on the network including clients. This removes the affected proxy server from the topological lists and de-peers network tunnel connections as well as deleting cached certificate of the device on the ADS. The proxy server will need to register with the ADS to rejoin the network once it has regained operational stability.

Proxy servers do not keep logs of any kind, not even error logs. When an error happens on a proxy server, the error report is sent to the domain controller, the ADS is the only device on the network that keeps a log. Log entries written to the ADS log file are constricted to the time, device name, and the error message, no information about clients or traffic flows are logged. The ADS log file is encrypted on the hard drive, requires administrator privileges to access, and can only be read from the command console that administers over the ADS server. All servers on the network are password protected, not only at the operating system level, but by the administrative consoles used to configure the servers. Files that are written to disk like configurations, topology, and log files, are encrypted by default, and decrypted temporarily only as they are loaded into memory for access.

Other types of error conditions may occur, for example when setting up the network. Failures with key exchanges, signing calls, and other administrative functions are echoed to the console display, and are meant to be handled by the administrator of the device during the setup phase of the network. These errors may be logged to the ADS, and can be reviewed by the administrator using built-in functions like network, certificate, topology, and logging commands available at the console level.

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_mek_exchange_request()*
- *aern_network_mek_exchange_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:

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

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

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.

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

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:

$\sigma(H(C_D^{\sigma} \parallel st)) = Sign_{Kpri}(H(C_D^{\sigma} \parallel st))$

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

$Verify_{rootKpub}(\sigma(H(C_D))) = H(C_D)$

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

$Verify_{devKpub}(\sigma(H(C_D^{\sigma} \parallel st))) = H(C_D^{\sigma} \parallel st)$

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, where a shared secret is exchanged between devices. 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_network_mek_exchange_request()*

- *aern_network_mek_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 ADS 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.

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

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

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

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.

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

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

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.

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

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

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

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

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:

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

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 $H(m \parallel ts)$ 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:

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

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

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

$Verify_{rootKpub}(\sigma(H(C_D)) = H(C_D)$

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

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.

$H_{CLS}^{\sigma} = Sign_{adcKpri}(H(C_D^{\sigma} \parallel list \parallel st))$

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

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*.

$Verify_{rootKpub}(\sigma(H(C_D)) = H(C_D)$

$Verify_{adcKpub}(H_{CLS}^{\sigma}) = H(C_D^{\sigma} \parallel list \parallel st)$

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:

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

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

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

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

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

The remote signing response:

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

Response(C_D^{σ})

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

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

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:

$H_{SS}^{\sigma} = Sign_{devKpri}(H(S_D \parallel st))$

$Request(S_D) = (S_D \parallel H_{SS}^{\sigma})$

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

$Verify_{devKpub}(H_{SS}^{\sigma}) = H(S_D \parallel st)$

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 subsect 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.

$H_{SS}^{\sigma} = Sign_{dlaKpri}(H(S_D \,||\, st))$

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

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

$L = \{ D_1, D_2, \ldots, D_n \}$

For each $i \in L = Broadcast(L_i, (S_D \,\|\, \sigma))$

6.8 Join Request

Overview:

Before a Client contacts an ADS 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 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:

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

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

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.

$Verify_{rootKpub}(\sigma(H(C_D)) = H(C_D)$

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

The registration response:

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

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

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

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.

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

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

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.

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

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

7. Security Analysis

AERN is designed to provide robust security against a wide range of attacks, including classical and quantum threats. The protocol incorporates multiple layers of security measures to ensure confidentiality, integrity, authentication, and forward secrecy.

7.1 Introduction to Security Context

The Authenticated Encrypted Relay Network (AERN) is designed as an advanced anonymizing protocol for secure, anonymous communications, explicitly addressing vulnerabilities identified in prior anonymizing network implementations such as The Onion Router (TOR). This analysis compares AERN's security architecture to TOR and similar anonymizing systems, highlighting the security strengths inherent in AERN's fully private, authenticated network approach.

7.2 Comparative Analysis with TOR

TOR, while innovative and widely adopted, operates as a partially public network in which volunteer-run nodes introduce inherent trust and security vulnerabilities. Nodes may be compromised by malicious actors, leading to attacks such as exit node monitoring, Sybil attacks, timing correlation, and targeted metadata harvesting. Furthermore, TOR's reliance on layered asymmetric cryptography at each hop incurs significant computational overhead, limiting scalability and increasing latency.

AERN addresses these concerns through several key architectural improvements:

- **Fully Authenticated Private Infrastructure:** Unlike TOR, AERN nodes are fully controlled and authenticated via a robust certificate-based trust model. The ARS and ADC provide rigorous oversight of proxy server membership, significantly reducing the potential for malicious node infiltration.
- **Post-Quantum Cryptographic Security:** Utilizing quantum-resistant cryptographic primitives such as Kyber, McEliece, Dilithium, and SPHINCS+, AERN ensures longevity and resistance to cryptographic attacks, particularly those anticipated with quantum computing advancements. In contrast, TOR remains susceptible to quantum attacks due to its reliance on classical cryptography.

- **Symmetric Encryption Tunneling (RCS Cipher):** By employing pre-negotiated symmetric encryption tunnels between nodes, AERN achieves computational efficiency and lower latency. This efficiency enables longer proxy chains (up to 16 nodes or more) without significantly impacting performance, thus enhancing anonymity by increasing complexity and obscurity in traffic patterns.
- **Dynamic Randomized Routing:** Each packet traverses a newly randomized path across the proxy network. This strategy prevents attackers from reliably conducting traffic correlation attacks, a significant vulnerability in the fixed routing model typically used by TOR circuits.

7.3 Security Strengths in Private Implementation

The closed nature of the AERN network, combined with stringent authentication and comprehensive encryption protocols, delivers enhanced security over public or semi-public networks. Key strengths include:

- **Elimination of Malicious Node Risk:** Since all network devices are authenticated by the ARS and ADC, malicious nodes (prevalent in public anonymizing networks) are effectively eliminated. The enforced topological integrity ensures node trustworthiness and significantly diminishes the probability of insider threats.
- **Metadata Resistance and Traffic Obfuscation:** Standardized packet sizes and constant-time encryption processes render packet analysis attacks ineffective. The use of authenticated stream cipher encryption (RCS), coupled with per-hop re-encryption, obscures traffic patterns and provides robust metadata protection.
- **Replay and Timing Attack Mitigation:** The protocol integrates timestamps and sequence numbers, cryptographically verified at each hop, effectively mitigating replay attacks and timing-based correlation attacks.
- **Robust Certificate-Based Authentication:** Each device's certificate, verified and signed by the ARS trust anchor, creates a hierarchical chain of trust. This prevents unauthorized devices from joining or injecting malicious traffic into the network.

7.4 Cryptanalytic Considerations

AERN's security model is resilient against classical cryptanalytic methods such as ciphertext analysis, brute-force attacks, and side-channel exploits due to its advanced symmetric cipher (RCS) and quantum-resistant asymmetric primitives. Post-quantum cryptographic algorithms further secure AERN against future quantum computing threats, addressing vulnerabilities unmitigated by current-generation anonymizing networks.

7.5 Potential Limitations and Mitigations

- **Trust Centralization:** AERN's hierarchical certificate authority (ARS and ADC) may present a centralized trust risk. To mitigate this, the ARS is deliberately isolated, and signing operations can be proxied via the ADC. This minimizes potential single points of failure or compromise.
- **Resource Scalability:** Each node maintains symmetric encryption tunnels with other nodes, posing potential scalability concerns. AERN mitigates this through computationally efficient encryption and key caching mechanisms, optimized for large-scale implementations.

7.6 Conclusion of Security Analysis

AERN presents a significant advancement over existing anonymizing networks by offering a secure, authenticated, and robust private relay network designed to address TOR's documented vulnerabilities. Its cryptographic framework ensures resilience against current and future threats, while its private implementation paradigm ensures complete control over infrastructure and node trustworthiness. AERN stands as a viable, secure alternative to TOR for sensitive and secure communications, demonstrating substantial improvements in both security and practical efficiency.

8. Conclusion

Strengths and Innovations of AERN:

1. Authenticated and Controlled Infrastructure:

- TOR's primary vulnerability stems from its open infrastructure. Volunteer-operated nodes expose the network to malicious actors. AERN addresses this explicitly by enforcing strict authentication and certificate management through its Root Security Server (**ARS**) and Domain Controller (**ADC**). This centralized trust and authentication model significantly mitigates threats such as malicious exit nodes and compromised directory servers.

2. Quantum-Resistant Cryptography:

- AERN explicitly integrates quantum-resistant cryptographic primitives (**Kyber, McEliece, Dilithium, SPHINCS+**), providing robust security against future quantum-computing threats. This is an area where TOR and most conventional VPN protocols (which rely on classical cryptography like RSA, ECC, and Diffie-Hellman) are significantly weaker, being inherently vulnerable to future quantum attacks.

3. Dynamic Per-Packet Randomized Routing:

- Unlike TOR's fixed-path circuits (three-node structure maintained for roughly ten minutes), AERN employs a randomized path for **every packet**, creating substantial difficulty in traffic analysis and correlation attacks. This is a significant security improvement, substantially increasing the complexity required for successful adversarial traffic analysis.

4. Efficient Symmetric Cipher Utilization (RCS):

- The use of pre-established, authenticated symmetric cipher tunnels provides both speed and security. Compared to TOR's layered encryption model, which incurs significant computational latency at each node, the symmetric cipher (RCS) in AERN significantly improves latency and scalability. This allows for more extensive routing chains, enhancing overall anonymity without sacrificing performance.

5. Robust Metadata Protection:

- Standardized packet sizes and the encrypted tunnel system prevent metadata harvesting, packet-size analysis, and timing attacks. Such consistent and rigorous metadata protection

is a notable advancement over typical VPN and TOR implementations, where metadata leakage remains an ongoing challenge.

6. Replay and Timing Attack Mitigation:

AERN integrates packet timestamping, sequencing, and cryptographic verification at each hop, explicitly designed to mitigate replay attacks, timing attacks, and packet manipulation attempts. TOR and standard VPNs typically lack similarly rigorous mechanisms at each routing step.

Potential Limitations and Considerations:

1. Centralized Trust Model:

The primary security trade-off of AERN's design is its reliance on centralized trust management via ARS and ADC. Although explicitly designed to mitigate risks, any compromise or operational failure in these centralized components could present significant vulnerabilities. AERN proactively mitigates this risk by strictly isolating the ARS from general network interactions, limiting its exposure.

2. Complexity of Implementation:

The rigorous cryptographic and administrative design of AERN means that complexity and precision in implementation are crucial. Any misconfiguration or errors in the cryptographic implementation could introduce subtle vulnerabilities. Rigorous auditing and certification of the cryptographic implementations would be necessary to ensure optimal security.

Comparative Verdict (AERN vs. TOR and VPNs):

- **Security:** AERN offers significant cryptographic and structural security improvements over TOR and typical VPN technologies. It explicitly addresses TOR's weaknesses related to malicious nodes, quantum vulnerability, and traffic analysis.
- **Performance:** By leveraging symmetric encryption after initial setup, AERN provides improved performance compared to TOR's layered asymmetric encryption approach, making it well-suited for latency-sensitive applications like VoIP or streaming.
- **Future-Proofing:** AERN's explicit focus on quantum-resistant cryptography gives it a substantial advantage in long-term security planning compared to traditional systems still relying on classical cryptographic primitives.
- **Scalability and Efficiency:** Despite the sophisticated security features, AERN maintains scalability through computationally efficient encryption strategies and node management.

Summary:

From a cryptographic and network security perspective, AERN is objectively a significant improvement on TOR and conventional anonymizing VPNs. It proactively addresses key known vulnerabilities, significantly improves cryptographic strength (especially in terms of quantum resistance), reduces susceptibility to traffic correlation, and substantially enhances performance.

While the centralized trust model introduces specific management risks, these are explicitly mitigated through careful architectural choices.

In short, AERN is a well-designed, forward-looking protocol that substantially enhances anonymizing network security beyond what TOR and traditional VPN solutions currently provide.