KERI specification

Create technical specifications in markdown. Based on the original Spec-Up, extended with Terminology tooling

---

CONTENTS

- Key Event Receipt Infrastructure (KERI)
  - Introduction
  - Status of This Memo
  - Copyright Notice
  - Terms of Use
  - Scope
  - Normative references
  - Terms and Definitions
  - KERI foundational overview
    - Infrastructure and ecosystem overview
      - Controller Application
      - Direct exchange
      - Indirect exchange via witnesses and watchers
      - Ecosystem
    - KERI's identifier system security overlay
    - Overcoming existing security overlay flaws
    - End-verifiable
    - Self-certifying identifier (SCID)
    - Autonomic identifier (AID)
    - Key rotation/pre-rotation
    - Qualified Cryptographic Primitives
    - CESR Encoding
    - KERI's secure bindings
      - Tetrad bindings

Autonomic Namespaces (ANs)

KERI data structures and labels

KERI data structure format

KERI field labels for data structures

Compact KERI field labels

Special label ordering requirements

Version string field

Legacy version string field format

Message type field

SAID fields

AID fields

Sequence number field

Key and key digest threshold fields

Key list field

Next key digest list field

Backer threshold field

Backer list

Backer remove list

Backer add list

Configuration traits field

Seal list field

Seals

Seal Count Codes

Digest seal

Merkle Tree root digest seal

Source Event seal

- Key Event seal
- Latest establishment event seal
- Registrar Backer seal
- Typed seal
- Key event messages
  - Inception Event Message Body
    - Inception Event Example
  - Delegated Inception Event Message Body
  - Interaction Event Message Body
    - Interaction Event Example
  - Rotation Event Message Body
    - Rotation Event Example
  - Delegated Inception Event Message Body
    - Delegated Inception Event Example
  - Delegated Rotation Event Message Body
    - Delegated Rotation Event Example
- Receipt Messages
  - Receipt Message Body
    - Receipt example:
- Routed Messages
  - Routed Services
  - Routing Security
  - Reserved field labels in routed messages
    - UUID Salty Nonce field
    - Controller AID field
    - Receiver AID field

- Prior event SAID field
- Exchange identifier field
- Datetime, dt field
- Route field
- Return Route field
- Query field
- Attribute field
- Query Message Body
  - Example Query Message
- Reply Message Body
  - Example Reply Message
- Prod Message Body
  - Example Prod Message
- Bare Message Body
  - Example Bare Message
- Exchange Transaction Inception Message Body
  - Example Exchange Transaction Inception Message
- Exchange Message Body
  - Example Exchange Transaction Message
- Signing and sealing KERI data structures
  - Indexed Signatures
  - Non-indexed signatures
  - Endorsements
  - Sealing
  - Receipt Signatures
  - Receipt Seals

KERI key management
KERI keypair labeling convention
Case in which only one index is needed
Case in which both indexes are needed
Labelling the digest of the public key
Labelling key events in a KEL
Pre-rotation
Inception event pre-rotation
Rotation using pre-rotation
Fractionally weighted threshold
General Pre-rotation
Reserve Rotation
Custodial Rotation
Surprise Quantum Attack Recovery (SQAR)
Cooperative Delegation
Security Properties of Pre-rotation
Dead-Attacks
Non-establishment Dead-attack
Establishment Dead-attack
Live-Attacks
Non-establishment Live-attack
Establishment Live-attack
Delegated Event Live-attacks
Annex A
Cryptographic strength and security
Cryptographic strength

Information theoretic security and perfect-security
Post-Quantum Security
KERI Security Properties
Validation
Verifier
Validator
Duplicity
Event Types and Classes
Validator Roles and Event Locality
Validation Rules
Superseding Recovery and Reconciliation
First Seen Policy
Reconciliation
Superseding Recovery
Superseding Rules for Recovery at a given location, SN (se…
KERI's Algorithm for Witness Agreement (KAWA)
Introduction
Advantages
Witness Designation
Witnessing Policy
Immunity and Availability
Security Properties
Working Examples Setup
AIDs
UUIDs
Native CESR Encodings of KERI Messages

- CESR Field Encodings
  - Protocol and Genus Version
  - DateTime
  - Threshold
  - Route or Return Route
- Key Event Messages
  - Inception icp
  - Interaction ixn
  - Delegated Inception dip
  - Rotation rot
  - Delegated Rotation drt
- Receipt Messages
  - Receipt rct
- KERI Routed Messages
- Query Message
- Reply Message
- Prod Message
- Bare Message
- Exchange Transaction Inception Message
- Exchange Message

Out-Of-Band-Introduction (OOBI)
- Basic OOBI
- OOBI URL (IURL)
- Well-Known OOBI
- CID and EID
- Multi-OOBI (MOOBI)

KERI Reply Messages as OOBIs
Self (Blind) OOBI (SOOBI)
OOBI Forwarding
OOBI with MFA
SPED (Speedy Percolated Endpoint Discovery)
JIT/NTK Discovery
Summary
BADA (Best-Available-Data-Acceptance) Policy
Security Issues
BADA Rules
KEL Anchored Updates
Signed (Not Anchored) Updates
RUN off the CRUD
OOBI KERI Endpoint Authorization (OKEA)
Authorized Endpoint Disclosure Example
Player EID in Role by CID Update
Player EID in Role by CID Nullify Via Cut
Endpoint Location with Scheme by EID Update
Endpoint Location with Scheme by EID Nullify Via Empty URL
Bibliography
Normative section
Informative section
Issues
Settings

Key Event Receipt Infrastructure (KERI)

Specification Status: v1.1

DOI

https://doi.org/10.5281/zenodo.18887102

Latest Draft:

https://github.com/trustoverip/kswg-keri-specification

Author:

- Samuel Smith, Prosapien

Editors:

- Kevin Griffin, GLEIF

Contributors:

- Samuel Smith, Prosapien
- Phil Feairheller, HealthKERI
- Kevin Griffin, GLEIF
- Nuttawut Kongsuwan
- Daniel Hardman
- Ed Edeykholt
- Sai Ranjit Tummalapalli
- Vasiliy Suvorov
- Charles Lanahan
- Fintan Halpenny
- Henk van Cann, Blockchainbird
- Kor Dwarshuis, Blockchainbird

Participate:

GitHub repo

Commit history

Introduction

The original design of the Internet Protocol (IP) has no security layer(s) [RFC0791], providing no built-in mechanism for secure attribution to the source of an IP packet. Anyone can forge an IP packet, and a recipient may not be able to ascertain when or if the packet was sent by an imposter. This means that secure attribution mechanisms for the Internet must be overlaid. This documents presents an identifier system security overlay, called the Key Event Receipt Infrastructure (KERI) protocol, that serves as a trust spanning layer for the Internet. This overlay includes a primary root-of-trust in a Self-certifying identifier (SCID) that provides a formalism for Autonomic identifiers (AID s), Autonomic namespaces (AN s), and the basis for a universal Autonomic identity system (AIS).

The KERI protocol provides verifiable authorship (authenticity) of any message or data item via secure cryptographically verifiable attribution to a SCID as a primary root-of-trust [4] [16] [18] [19] [17] [15]. This root-of-trust is cryptographic, not administrative, because it does not rely on any trusted third-party administrative process but may be established with cryptographically verifiable data structures. This cryptographic root-of-trust enables end verifiability where every data item may be cryptographically attributable to its source by any recipient verifier, without reliance on any infrastructure not under the verifier's ultimate control. Therefore, KERI has no security dependency on any other infrastructure and does not rely on security guarantees that may or may not be provided by the traditional internet infrastructure. This makes intervening operational infrastructure replaceable, enabling ambient verifiability (verification by anyone, anywhere, at any time).

A SCID is strongly bound at inception to a cryptographic keypair that is self-contained unless control over the SCID needs to be transferred to a new keypair. The KERI protocol provides end-verifiable control provenance over a variant of SCID, called an Autonomic identifier (AID), via signed transfer statements in an append-only chained Key event log (KEL). The key management operation for tansferring control over an AID is implemented via a novel key pre-rotation scheme [6]. With pre-rotation, control over an AID can be re-established by rotating to a one-time use set of unexposed but pre-committed rotation keypairs. This approach fixes the foundational flaw in traditional Public key infrastructure (PKI), which is insecure key rotation. KERI enables decentralized public key infrastructure (DPKI) that is more secure and portable. KERI may be viewed as a viable reboot of the Web-of-Trust concept for DPKI because KERI fixes the hard problem of DPKI, which is key rotation.

Two primary trust modalities motivated the design of the KERI protocol, namely a direct (one-to-one) mode and an indirect (one-to-any) mode. In the direct mode, two entities establish trust over AIDs via a direct exchange of their counterparts' verified signatures. In the indirect mode, trust over AIDs depends on witnessed Key event receipt logs (KERL s, Key event receipt) as a secondary root-of-trust for validating key events. The security and accountability guarantees of indirect mode are provided by KERI's Algorithm for Witness Agreement (KAWA) among a set of key event Witnesses. The KAWA ap-

proach may be much more performant and scalable than more complex approaches that depend on a total ordering distributed consensus ledger. Nevertheless, KERI may employ a distributed consensus ledger when other considerations make it the best choice.

The KERI approach to Decentralized key management infrastructure (DKMI) allows for more granular composition. Moreover, because KERI is event streamed, it enables DKMI to operate in-stride with data events streaming applications such as web 3.0, IoT, and others where performance and scalability are more important. The core KERI engine is independent of identifier namespace. This makes KERI a candidate for a universal portable DKMI. This system uses the design principle of minimally sufficient means for appropriate levels of security, performance, and adoptability to be a viable candidate as the DKMI that underpins a trust-spanning layer for the Internet.

Status of This Memo

Information about the current status of this document, any errata, and how to provide feedback on it, may be obtained at https://github.com/trustoverip/kswg-keri-specification.

Copyright Notice

This specification is subject to the **OWF Contributor License Agreement 1.0 - Copyright** available at https://www.openwebfoundation.org/the-agreements/the-owf-1-0-agreements-granted-claims/owf-contributor-license-agreement-1-0-copyright.

If source code is included in the specification, that code is subject to the Apache 2.0 license unless otherwise marked. In the case of any conflict or confusion between the OWF Contributor License and the designated source code license within this specification, the terms of the OWF Contributor License MUST apply.

These terms are inherited from the Technical Stack Working Group at the Trust over IP Foundation. Working Group Charter.

Terms of Use

These materials are made available under and are subject to the OWF CLA 1.0 - Copyright & Patent license. Any source code is made available under the Apache 2.0 license.

THESE MATERIALS ARE PROVIDED “AS IS.” The Trust Over IP Foundation, established as the Joint Development Foundation Projects, LLC, Trust Over IP Foundation Series (“ToIP”), and its members and contributors (each of ToIP, its members and contributors, a “ToIP Party”) expressly disclaim any warranties (express, implied, or otherwise), including implied warranties of merchantability, non-infringement, fitness for a particular

purpose, or title, related to the materials. The entire risk as to implementing or otherwise using the materials is assumed by the implementer and user. IN NO EVENT WILL ANY ToIP PARTY BE LIABLE TO ANY OTHER PARTY FOR LOST PROFITS OR ANY FORM OF INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER FROM ANY CAUSES OF ACTION OF ANY KIND WITH RESPECT TO THESE MATERIALS, ANY DELIVERABLE OR THE ToIP GOVERNING AGREEMENT, WHETHER BASED ON BREACH OF CONTRACT, TORT (INCLUDING NEGLIGENCE), OR OTHERWISE, AND WHETHER OR NOT THE OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Scope

Implementation design of a protocol-based decentralized key management infrastructure that enables secure attribution of data to a cryptographically derived identifier with strong (cryptographically verifiable) bindings between each of the identifier, a set of asymmetric signing key pairs that are the key state, a controlling entity that holds the private keys, and a cryptographically verifiable data structure that enables changes to that key state. Thus, security over secure attribution is reduced to key management. This key management includes, for the first time, a practical solution to the hard problem of public key rotation. There is no reliance on trusted third parties. The resulting secure attribution is fully end-to-end verifiable.

Because of the reliance on asymmetric (public, private) digital signing key pairs, this may be viewed as a type of decentralized public key infrastructure (DPKI). The protocol supports cryptographic agility for both pre- and post-quantum attack resistance. The application scope includes any electronically transmitted information. The implementation dependency scope assumes no more than cryptographic libraries that provide cryptographic strength pseudo-random number generators, cryptographic strength digest algorithms, and cryptographic strength digital Signature algorithms.

Normative references

The normative documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 7498-1:1994 Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model, including Requirement Levels (IETF RFC-2119).

See Bibliography - Normative Section

Terms and Definitions

For the purposes of this document, the following terms and definitions apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

- ISO Online browsing platform: available at https://www.iso.org/obp
- IEC Electropedia: available at http://www.electropedia.org/

. **Indexed signature** (*indexed-signature*)

Property	Value
Original Term	indexed-signature
Link	https://trustoverip.github.io/kerisuite-glossary/#term:indexed-signature
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

Also called *siger*. An indexed signature attachment is used when signing anything with a multi-key autonomic identifier. The index is included as part of the attachment, so a verifier knows which of the multiple public keys was used to generate a specific signature.

Source:Philip Feairheller

More in extended KERI glossary

Authentic Chained Data Container (ACDC, *authentic-chained-data-container*)

Property	Value
Original Term	authentic-chained-data-container
Link	https://trustoverip.github.io/kerisuite-glossary/#term:authentic-chained-data-container
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a directed acyclic graph with properties to provide a verifiable chain of proof-of-authorship. See the full specification

Source: Dr. S.Smith, 2024

Explained briefly, an ACDC or authentic-data-container proves digital data consistency and authenticity in one go. An ACDC cryptographically secures commitment to the data contained, and its identifiers are self-addressing, which means they point to themselves and are also contained in the data.

More in extended KERI glossary

Autonomic identifier (AID, *autonomic-identifier*)

Property	Value
Original Term	autonomic-identifier
Link	https://trustoverip.github.io/kerisuite-glossary/#term:autonomic-identifier
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a self-managing cryptonymous identifier that must be self-certifying (self-authenticating) and must be encoded in CESR as a qualified Cryptographic primitive.

Source: Dr. S.Smith, 2024

An identifier that is self-certifying-identifier and self-sovereign-identity (or *self-managing*).

More in extended KERI glossary

➪ **Autonomic identity system** (AIS, *autonomic-identity-system*)

Property	Value
Original Term	autonomic-identity-system
Link	https://trustoverip.github.io/ctwg-main-glossary/#term:autonomic-identity-system
Owner	trustoverip
Repo	ctwg-main-glossary
Commit hash	00608bb6ce7f7e6285f202b580548d61ee716621

an identity system that includes a primary root-of-trust in self-certifying identifiers that are strongly bound at issuance to a cryptographic signing (public, private) key pair. An AIS enables any entity to establish control over an AN in an independent, interoperable, and portable way.

Source: Dr. S.Smith, 2024

There's nobody that can intervene with the establishment of the authenticity of a control operation because you can verify all the way back to the root-of-trust.

➪ **Autonomic namespace** (AN, *autonomic-namespace*)

Property	Value
Original Term	autonomic-namespace
Link	https://trustoverip.github.io/ctwg-main-glossary/#term:autonomic-namespace
Owner	trustoverip
Repo	ctwg-main-glossary
Commit hash	00608bb6ce7f7e6285f202b580548d61ee716621

a namespace that is self-certifying and hence self-administrating. An AN has a self-certifying prefix that provides cryptographic verification of root control authority over its namespace. All derived AIDs in the same AN share the same root-of-trust, source-of-truth, and locus-of-control (RSL). The governance of the namespace is, therefore, unified into one entity, that is, the controller who is/holds the root authority over the namespace.

Source: Dr. S.Smith, 2024

Namespaces are, therefore, portable and truly self-sovereign.

Backer (*backer*)

Property	Value
Original Term	backer
Link	https://trustoverip.github.io/kerisuite-glossary/#term:backer
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

an alternative to a traditional KERI based Witness commonly using Distributed Ledger Technology (DLT) to store the KEL for an identifier.

Source: Dr. S.Smith, 2024

More in extended KERI glossary

Concise Binary Object Representation (CBOR, *concise-binary-object-repres...*)

Property	Value
Original Term	concise-binary-object-representation
Link	https://trustoverip.github.io/ctwg-general-glossary/#term:concise-binary-object-representation
Owner	trustoverip
Repo	ctwg-general-glossary
Commit hash	73604cf54c9f238b0d239daecc413a045f61c75e

It is a binary data serialization format loosely based on JSON authored by C. Bormann. Like JSON it allows the transmission of data objects that contain name–value pairs, but in a more concise manner. This increases processing and transfer speeds at the cost of human readability.

Also knows as: CBOR

⇨ **Configuration traits** (Modes, *configuration-traits*)

Property	Value
Original Term	configuration-traits
Link	https://trustoverip.github.io/kerisuite-glossary/#term:configuration-traits
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a list of specially defined strings representing a configuration of a KEL.

Source: Dr. S.Smith, 2024

More in extended KERI glossary

See also: Configuration traits field.

⇨ **Controller** (*controller*)

Property	Value
Original Term	controller
Link	https://trustoverip.github.io/ctwg-main-glossary/#term:controller
Owner	trustoverip
Repo	ctwg-main-glossary
Commit hash	00608bb6ce7f7e6285f202b580548d61ee716621

In the context of digital communications, the entity in control of sending and receiving digital communications. In the context of decentralized digital trust infrastructure, the entity in control of the cryptographic keys necessary to perform cryptographically verifiable actions using a digital agent and digital wallet. In a ToIP context, the entity in control of a ToIP endpoint.

See also: device controller, DID controller, ToIP controller.

Supporting definitions:

eSSIF-Lab: the role that an actor performs as it is executing actions on that entity for the purpose of ensuring that the entity will act/behave, or be used, in a particular way.

Cryptographic primitive (Cryptographic primitives, *cryptographic-primitive*)

Property	Value
Original Term	cryptographic-primitive
Link	https://trustoverip.github.io/ctwg-general-glossary/#term:cryptographic-primitive
Owner	trustoverip
Repo	ctwg-general-glossary
Commit hash	73604cf54c9f238b0d239daecc413a045f61c75e

the serialization of a value associated with a cryptographic operation including but not limited to a digest (hash), a salt, a seed, a private key, a public key, or a signature.

Source: Dr. S.Smith, 2024

Cryptonym (Cryptonymous, *cryptonym*)

Property	Value
Original Term	cryptonym
Link	https://trustoverip.github.io/ctwg-general-glossary/#term:cryptonym
Owner	trustoverip
Repo	ctwg-general-glossary
Commit hash	73604cf54c9f238b0d239daecc413a045f61c75e

a cryptographic pseudonymous identifier represented by a string of characters derived from a random or pseudo-random secret seed or salt via a one-way cryptographic function with a sufficiently high degree of cryptographic strength (e.g., 128 bits, see appendix on cryptographic-strength. A cryptonym is a type of primitive .

Current threshold (*current-threshold*)

Property	Value
Original Term	current-threshold
Link	https://trustoverip.github.io/kerisuite-glossary/#term:current-threshold
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

represents the number or fractional weights of signatures from the given set of current keys required to be attached to a Message for the Message to be considered fully signed.

Source: Dr. S.Smith, 2024

More in extended KERI glossary

Dead-attack (*dead-attack*)

Property	Value
Original Term	dead-attack
Link	https://trustoverip.github.io/kerisuite-glossary/#term:dead-attack
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

an attack on an establishment-event that occurs after the Key-state for that event has become stale because a later establishment event has rotated the sets of signing and pre-rotated keys to new sets.

More in extended KERI glossary

⇨ **Decentralized Key Management Infrastructure** (DKMI, *decentralized-key-m…*

Property	Value
Original Term	decentralized-key-management-infrastructure
Link	https://trustoverip.github.io/ctwg-general-glossary/#term:decentralized-key-management-infrastructure
Owner	trustoverip
Repo	ctwg-general-glossary
Commit hash	73604cf54c9f238b0d239daecc413a045f61c75e

a key management infrastructure that does not rely on a single entity for the integrity and security of the system as a whole. Trust in a DKMI is decentralized through the use of technologies that make it possible for geographically and politically disparate entities to reach an agreement on the key state of an identifier DPKI.

Source: Dr. S.Smith, 2024

⇨ **Duplicity** (*duplicity*)

Property	Value
Original Term	duplicity
Link	https://trustoverip.github.io/kerisuite-glossary/#term:duplicity
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

the existence of more than one version of a Verifiable key-event-log for a given AID.

Source: Dr. S.Smith, 2024

More in extended KERI glossary

⇨ **End-verifiability** (*end-verifiability*)

Property	Value
Original Term	end-verifiability
Link	https://trustoverip.github.io/kerisuite-glossary/#term:end-verifiability
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a data item or statement may be cryptographically securely attributable to its source (party at the source end) by any recipient verifier (party at the destination end) without reliance on any infrastructure not under the verifier’s ultimate control.

Source Dr. S.Smith

⇨ **Establishment event** (*establishment-event*)

Property	Value
Original Term	establishment-event
Link	https://trustoverip.github.io/kerisuite-glossary/#term:establishment-event
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a key-event that establishes or changes the key state which includes the current set of authoritative keypairs (key state) for an AID .

Source: dr. S.Smith

More in extended KERI glossary

⇨ **Exchange identifier** (*exchange-identifier*)

Property	Value
Original Term	exchange-identifier
Link	https://trustoverip.github.io/kerisuite-glossary/#term:exchange-identifier
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a cryptographically verifiable identifier used to represent a party in a data exchange. It facilitates secure attribution (authenticity), content protection (confidentiality), and may support privacy via uncorrelatable or pseudonymous metadata.

⇨ **First-seen** (First seen, *first-seen*)

Property	Value
Original Term	first-seen
Link	https://trustoverip.github.io/kerisuite-glossary/#term:first-seen
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

refers to the first instance of a message received by any witness or watcher. The first-seen event is always seen, and can never be unseen. It forms the basis for duplicity detection in KERI-based systems.

Source: Dr. S.Smith

More in extended KERI glossary

Inception (*inception*)

Property	Value
Original Term	inception
Link	https://trustoverip.github.io/kerisuite-glossary/#term:inception
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

The operation of creating an AID by binding it to the initial set of authoritative keypairs and any other associated information. This operation is made verifiable and duplicity evident upon acceptance as the inception event that begins the AID's KEL.

Source Sam Smith

More in extended KERI glossary

Inception event (*inception-event*)

Property	Value
Original Term	inception-event
Link	https://trustoverip.github.io/kerisuite-glossary/#term:inception-event
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

an establishment-event that provides the incepting information needed to derive an AID and establish its initial Key state.

Source Sam Smith

More in extended KERI glossary

⇨ **Interaction event** (*interaction-event*)

Property	Value
Original Term	interaction-event
Link	https://trustoverip.github.io/kerisuite-glossary/#term:interaction-event
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

Non-establishment Event that anchors external data to the key-state as established by the most recent prior establishment event.

Source Sam Smith

More in extended KERI glossary

⇨ **KERI's Algorithm for Witness Agreement** (KAWA, *keris-algorithm-for-witnes…*

Property	Value
Original Term	keris-algorithm-for-witness-agreement
Link	https://trustoverip.github.io/kerisuite-glossary/#term:keris-algorithm-for-witness-agreement
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a type of Byzantine Fault Tolerant (byzantine-fault-tolerance) algorithm.

Source: Dr. S.Smith

More in extended KERI glossary

Key event (*key-event*)

Property	Value
Original Term	key-event
Link	https://trustoverip.github.io/kerisuite-glossary/#term:key-event
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

See the more general TrustoverIP concept of key-event: Key event

Concretely, it is the serialized data structure of an entry in the Key event log (KEL) for an [AID](autonomic-identifier). Abstractly, the data structure itself. Key events come in different types and are used primarily to establish or change the authoritative set of keypairs and/or anchor other data to the authoritative set of keypairs at the point in the KEL actualized by a particular entry.

Source Sam Smith

More in extended KERI glossary

Key event log (KEL, kel, *key-event-log*)

Property	Value
Original Term	key-event-log
Link	https://trustoverip.github.io/kerisuite-glossary/#term:key-event-log
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a Verifiable data structure that is a backward and forward chained, signed, append-only log of key events for an AID. The first entry in a KEL must be the one and only Inception event of that AID.

Source Sam Smith

More in extended KERI glossary

Key event message (*key-event-message*)

Property	Value
Original Term	key-event-message
Link	https://trustoverip.github.io/kerisuite-glossary/#term:key-event-message
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

Message whose body is a key event and whose attachments may include signatures on its body.

Source Sam Smith

More in extended KERI glossary

Key event receipt (*key-event-receipt*)

Property	Value
Original Term	key-event-receipt
Link	https://trustoverip.github.io/kerisuite-glossary/#term:key-event-receipt
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

message whose body references a Key event and whose attachments must include one or more signatures on that Key event.

Source Sam Smith

More in extended KERI glossary

⇨ **Key event receipt infrastructure** (KERI, *key-event-receipt-infrastructure*)

Property	Value
Original Term	key-event-receipt-infrastructure
Link	https://trustoverip.github.io/kerisuite-glossary/#term:key-event-receipt-infrastructure
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

or the KERI protocol, is an identity system-based secure overlay for the Internet.

Source: Dr. S.Smtih

More in extended KERI glossary

⇨ **Key event receipt log** (KERL, *key-event-receipt-log*)

Property	Value
Original Term	key-event-receipt-log
Link	https://trustoverip.github.io/kerisuite-glossary/#term:key-event-receipt-log
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a key event receipt log is a kel that also includes all the consistent key event receipt message s created by the associated set of witnesses.

Source: Dr. S.Smith

More in extended KERI glossary

See annex Key event receipt log.

Key-state (*key-state*)

Property	Value
Original Term	key-state
Link	https://trustoverip.github.io/ctwg-general-glossary/#term:key-state
Owner	trustoverip
Repo	ctwg-general-glossary
Commit hash	73604cf54c9f238b0d239daecc413a045f61c75e

a set of currently authoritative keypairs for an AID and any other information necessary to secure or establish control authority over an AID. This includes current keys, prior next key digests, current thresholds, prior next thresholds, witnesses, witness thresholds, and configurations.

Live-attack (Live-attacks, *live-attack*)

Property	Value
Original Term	live-attack
Link	https://trustoverip.github.io/kerisuite-glossary/#term:live-attack
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

an attack that compromises either the current signing keys used to sign non-establishment events or the current pre-rotated keys needed to sign a subsequent establishment event. See (Security Properties of Prerotation)[#live-attacks].

Source: Dr. S.Smith

More in extended KERI glossary

See Security Properties of Prerotation.

⇨ **Message** (*message*)

Property	Value
Original Term	message
Link	https://trustoverip.github.io/kerisuite-glossary/#term:message
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a serialized data structure that comprises its body and a set of serialized data structures that are its attachments. Attachments may include but are not limited to signatures on the body.

Source: Dr. S.Smith

Also see: message in ToIP main glossary.

⇨ **MsgPack** (MGPK, *messagepack*)

Property	Value
Original Term	messagepack
Link	https://trustoverip.github.io/ctwg-general-glossary/#term:messagepack
Owner	trustoverip
Repo	ctwg-general-glossary
Commit hash	73604cf54c9f238b0d239daecc413a045f61c75e

MessagePack is a computer data interchange format. It is a binary form for representing simple data structures like arrays and associative arrays. MessagePack aims to be as compact and simple as possible. The official implementation is available in a variety of languages

Next threshold (*next-threshold*)

Property	Value
Original Term	next-threshold
Link	https://trustoverip.github.io/kerisuite-glossary/#term:next-threshold
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

represents the number or fractional weights of signatures from the given set of next keys required to be attached to a Message for the Message to be considered fully signed.

More in extended KERI glossary

Non-establishment event (*non-establishment-event*)

Property	Value
Original Term	non-establishment-event
Link	https://trustoverip.github.io/kerisuite-glossary/#term:non-establishment-event
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a Key event that does not change the current Key state for an AID. Typically, the purpose of a Non-establishment event is to anchor external data to a given Key state as established by the most recent prior Establishment event for an AID.

Source: Dr. S. Smith

More in extended KERI glossary

➪ **Out-of-band introduction** (OOBI, *out-of-band-introduction*)

Property	Value
Original Term	out-of-band-introduction
Link	https://trustoverip.github.io/kerisuite-glossary/#term:out-of-band-introduction
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

Out-of-band Introductions (OOBIs) are discovery and validation of IP resources for key-event-receipt-infrastructure autonomic identifiers. **Discovery via URI, trust via KERI.**

The simplest form of a KERI OOBI is a namespaced string, a tuple, a mapping, a structured message, or structured attachment that contains both a KERI AID and a URL. The OOBI associates the URL with the AID.

More in extended KERI glossary

➪ **Primitive** (*primitive*)

Property	Value
Original Term	primitive
Link	https://trustoverip.github.io/ctwg-general-glossary/#term:primitive
Owner	trustoverip
Repo	ctwg-general-glossary
Commit hash	73604cf54c9f238b0d239daecc413a045f61c75e

a serialization of a unitary value. All Primitives in KERI must be expressed in composable-event-streaming-representation.

Source: Dr. S.Smith

Python dict (Python dictionary, *python-dict*)

A Python dict (short for dictionary) is a built-in data type for an associative array or hash table that stores key–value pairs: you look up a key and get its value.

Receipt (*receipt*)

Property	Value
Original Term	receipt
Link	https://trustoverip.github.io/kerisuite-glossary/#term:receipt
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

event message or reference with one or more witness signatures.

See Also:

key-event-receipt

More in extended KERI glossary

Receiver (*receiver*)

Property	Value
Original Term	receiver
Link	https://trustoverip.github.io/kerisuite-glossary/#term:receiver
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

the receiver (recipient) of an exchange message. The receiver is a controller on its associated KEL but is not the sender of the exchange message.

a receiver is a component that listens for and processes out-of-band messages, such as events or credentials, delivered via a mailbox system. It routes these messages to appropriate handlers for secure, asynchronous communication between KERI agents.

Rotation (*rotation*)

Property	Value
Original Term	rotation
Link	https://trustoverip.github.io/kerisuite-glossary/#term:rotation
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

The operation of revoking and replacing the set of authoritative key-pair for an AID . This operation is made verifiable and duplicity evident upon acceptance as a rotation event that is appended to the AID's KEL .

Source Sam Smith

More in extended KERI glossary

Rotation event (*rotation-event*)

Property	Value
Original Term	rotation-event
Link	https://trustoverip.github.io/kerisuite-glossary/#term:rotation-event
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

an Establishment Event that provides the information needed to change the Key state, which includes a change to the set of authoritative keypairs for an AID.

Source: Dr. S.Smith

More in extended KERI glossary

Routed exchange message (*routed-exchange-message*)

Property	Value
Original Term	routed-exchange-message
Link	https://trustoverip.github.io/kerisuite-glossary/#term:routed-exchange-message
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

is a KERI protocol message that allows one controller (sender) to transmit a confidential message to another controller (receiver) through one or more intermediaries (routers), while preserving authenticity (non-repudiation) and enabling confidentiality (through encryption), but potentially sacrificing privacy depending on routing visibility.

SAIDive (*saidive*)

any fields that are derived from digest of the message body using the SAID protocol.

Salt (*salt*)

Property	Value
Original Term	salt
Link	https://trustoverip.github.io/ctwg-general-glossary/#term:salt
Owner	trustoverip
Repo	ctwg-general-glossary
Commit hash	73604cf54c9f238b0d239daecc413a045f61c75e

random data fed as an additional input to a one-way function that hashes data.

Source: Dr. S. Smith

⇨ **Seal** (*seal*)

Property	Value
Original Term	seal
Link	https://trustoverip.github.io/ctwg-general-glossary/#term:seal
Owner	trustoverip
Repo	ctwg-general-glossary
Commit hash	73604cf54c9f238b0d239daecc413a045f61c75e

a seal is a cryptographic commitment in the form of a cryptographic digest or hash tree root (Merkle root) that anchors arbitrary data or a tree of hashes of arbitrary data to a particular event in the key event sequence.

Source: Dr. S. Smith

⇨ **Self-addressed data** (SAD, *self-addressed-data*)

Property	Value
Original Term	self-addressed-data
Link	https://trustoverip.github.io/kerisuite-glossary/#term:self-addressed-data
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a representation of data content from which a SAID is derived. The SAID is both cryptographically bound to (content-addressable) and encapsulated by (self-referential) its SAD said .

Source: Dr. S.Smith

More in extended KERI glossary

Self-addressing identifier (SAID, *self-addressing-identifier*)

Property	Value
Original Term	self-addressing-identifier
Link	https://trustoverip.github.io/kerisuite-glossary/#term:self-addressing-identifier
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

any identifier that is deterministically generated out of the content, or a digest of the content.

Source: Dr. S. Smtih

More in extended KERI glossary

Self-certifying identifier (SCID, *self-certifying-identifier*)

Property	Value
Original Term	self-certifying-identifier
Link	https://trustoverip.github.io/kerisuite-glossary/#term:self-certifying-identifier
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a type of Cryptonym that is uniquely cryptographically derived from the public key of an asymmetric signing keypair (public, private).

Source: Dr. S. Smith

Also see the TrustoverIP scope definition: Self-certifying identifier

More in extended KERI glossary

Signature (Signatures, *digital-signature*)

Property	Value
Original Term	digital-signature
Link	https://trustoverip.github.io/ctwg-main-glossary/#term:digital-signature
Owner	trustoverip
Repo	ctwg-main-glossary
Commit hash	00608bb6ce7f7e6285f202b580548d61ee716621

A digital signature is a mathematical scheme that uses cryptography for verifying the authenticity of digital messages or documents. A valid digital signature, where the prerequisites are satisfied, gives a recipient very high confidence that the message was created by a known sender (authenticity), and that the message was not altered in transit (integrity).

Source: Wikipedia.

Supporting definitions:

NIST-CSRC: The result of a cryptographic transformation of data which, when properly implemented, provides the services of: 1. origin authentication, 2. data integrity, and 3. signer non-repudiation.

Validator (*validator*)

Property	Value
Original Term	validator
Link	https://trustoverip.github.io/ctwg-general-glossary/#term:validator
Owner	trustoverip
Repo	ctwg-general-glossary
Commit hash	73604cf54c9f238b0d239daecc413a045f61c75e

any entity or agent that evaluates whether or not a given signed statement as attributed to an identifier is valid at the time of its issuance.

Source: Dr. S. Smith

Verifiable (*verifiable*)

Property	Value
Original Term	verifiable
Link	https://trustoverip.github.io/ctwg-main-glossary/#term:verifiable
Owner	trustoverip
Repo	ctwg-main-glossary
Commit hash	00608bb6ce7f7e6285f202b580548d61ee716621

In the context of digital communications infrastructure, the ability to determine the authenticity of a communication (e.g., sender, contents, claims, metadata, provenance), or the underlying sociotechnical infrastructure (e.g., governance, roles, policies, authorizations, certifications).

See also:appraisable, digital signature .

Verifier (*verifier*)

Property	Value
Original Term	verifier
Link	https://trustoverip.github.io/ctwg-main-glossary/#term:verifier
Owner	trustoverip
Repo	ctwg-main-glossary
Commit hash	00608bb6ce7f7e6285f202b580548d61ee716621

A role an agent performs to perform verification of one or more proofs of the claims in a digital credential or other verifiable data.

See also: relying party; issuer, holder.

Mental model: W3C Verifiable Credentials Data Model Roles & Information Flows

Supporting definitions:

W3C VC: A role an entity performs by receiving one or more verifiable credentials, optionally inside a verifiable presentation for processing. Other specifications might refer to this concept as a relying party.

eSSIF-Lab: a component that implements the capability to request peer agents to present (provide) data from credentials (of a specified kind, issued by specified parties), and to verify such responses (check structure, signatures, dates), according to its principal's verifier policy.

NIST The entity that verifies the authenticity of a digital signature using the public key.

Version (*version*)

Property	Value
Original Term	version
Link	https://trustoverip.github.io/kerisuite-glossary/#term:version
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

an instance of a KEL for an AID in which at least one event is unique between two instances of the KEL.

Source: Dr. S. Smith

More in extended KERI glossary

Watcher (*watcher*)

Property	Value
Original Term	watcher
Link	https://trustoverip.github.io/kerisuite-glossary/#term:watcher
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

an *entity* or *component* that keeps a copy of a kerl for an identifier but that is not designated by the *controller* of the identifier as one of its witnesses. See annex watcher.

Source: Dr. S.Smith

More in extended KERI glossary

Witness (*witness*)

Property	Value
Original Term	witness
Link	https://trustoverip.github.io/kerisuite-glossary/#term:witness
Owner	trustoverip
Repo	kerisuite-glossary
Commit hash	ab23372198bed5cda8bc0ac0289cead7a06d3320

a witness is an entity or component designated (trusted) by the controller of an identifier. The primary role of a witness is to verify, sign, and keep events associated with an identifier. A witness is the controller of its own self-referential identifier which may or may not be the same as the identifier to which it is a witness. See also keri's-algorithm-for-witness-agreement.

Source: Dr. S. Smith

More in extended KERI glossary

KERI foundational overview

Infrastructure and ecosystem overview

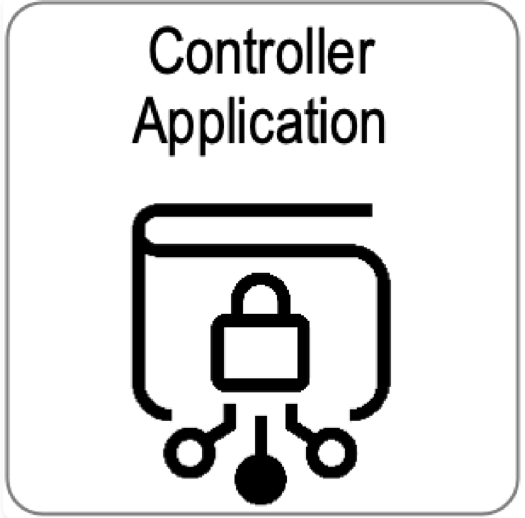
This section provides a high-level overview of the infrastructure components of a live KERI ecosystem and how they interact. It does not provide any low-level details and only describes the components superficially. However, it should help understand how all the parts fit together as one reads through the more detailed sections.

Controller Application

Each KERI AID is controlled by an entity (or entities when multi-sig) that holds the digital signing private keys belonging to the current authoritative key state of the AID. This set of entities is called the AID controller, or Controller for short. Each controller has an application or suite or applications called the controller application or application for short. The controller application provides five functions with respect to the digital signing key pairs that control the controller's AID. These five functions manage the associated key state via key events. These functions are:

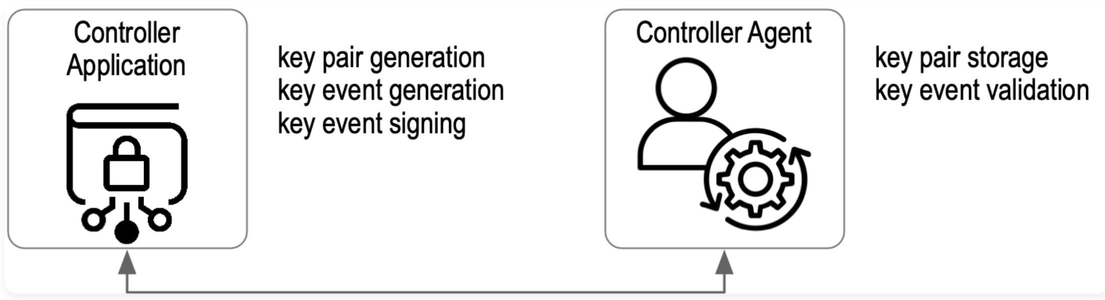
- key pair generation,
- key pair storage,
- key event generation,
- key event signing, and
- key event validation.

Key event validation includes everything needed to validate events, including structure validation, chaining digest verification, Signature verification, and Witness receipt verification. The execution of these functions, including the associated code and data, SHOULD be protected by the controller using best practices. For example, this might be accomplished by securely installing the controller application on a device in the physical possession of the controller, such as a mobile phone with appropriate secure storage and trusted code execution environments. Alternatively, the functions might be split between devices where a remote software agent that runs on behalf of the controller MAY host encrypted keypair storage and highly available key event validation functions while the more critical keypair generation, key event generation, and key event signing functions are on a device in the user's possession. The latter might be called a key chain or wallet. The extra security and scalability properties of delegated AIDs enable other arrangements for securely hosting the five functions. For the sake of clarity and without loss of generality, the controller application, including any devices or software agents, will be referred to as the controller application or application for short.



Controller Application

Figure: *Controller Application Functions*



Controller Application with Agent

Figure: *Controller Application with Agent*

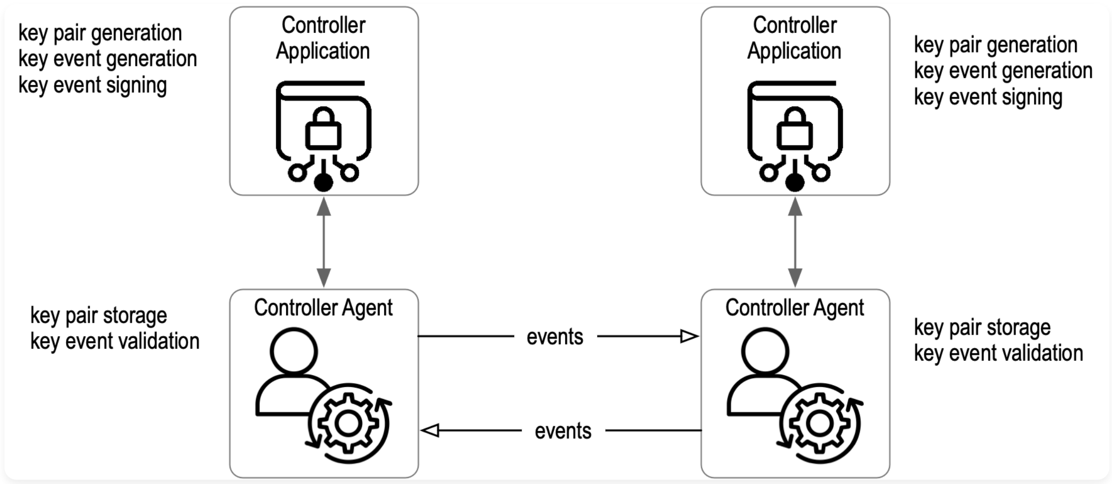
Direct exchange

The simplest mode of operation is that of a pair of controllers, each with their own AID, use their respective applications (including agents when applicable) to directly exchange Key event message s that verifiably establish the current key state of their own AID with the other controller. For each exchange of key events, the destination controller acts as a Validator of events received from the source controller. Therefore, given any key event, a given entity is either the event's controller or a validator of some other controller's event.

The set of key event messages forms an append-only, cryptographically Verifiable data structure called a Key event log or KEL. The events in a KEL are signed and are both forward and backward-chained. The backward chaining commitments are cryptographic digests of the previous event. The forward chaining commitments are cryptographic di-

gests of the next set of public keys that will constitute the key state after a key Rotation. The commitments are nonrepudiably signed with the private keys of the current key state. Each KEL is somewhat like a “blockchain” that manages the key state for one and only one AID. In addition to key states, each KEL also manages commitments to external data. These commitments are signed cryptographic digests of external data called seals. When included in a KEL, a seal binds (or anchors) the external data to the key state of the AID at the location in the KEL where the seal appears. This binding enables a controller to make cryptographically verifiable, nonrepudiable issuances of external data that are bound to a specific key state of that AID.

By exchanging KELs, each controller can validate the current key state of the other and, therefore, securely attribute (authenticate) any signed statements or any sealed issuances of data. This bootstraps the use of authentic data in any interaction or transaction between the pair of controllers. This is the mission of KERI.



Direct Exchange

Figure: *Direct Exchange*

Indirect exchange via witnesses and watchers

For many if not most use cases, the direct exchange of key event messages between controller applications (including agents when applicable) may not provide sufficient availability, scalability, or even security. KERI includes two other components for those use cases. These components are witnesses and Watchers.

Each controller of an AID MAY create or choose to use a set or pool of witnesses for that AID. The controller chooses how the witnesses are hosted. It MAY use a witness service provided by some other party or MAY directly host its own witnesses in its own infrastructure or some combination of the two. Regardless, the composition of the Witness pool is under the ultimate control of the AID’s controller, which means the controller MAY change the witness infrastructure at will. Witnesses for the AID are managed by the key

events in the AID's KEL. Each witness creates a signed Receipt of each event it witnesses, which is exchanged with the other witnesses (directly or indirectly). Based on those receipts, the witness pool uses an agreement algorithm called KAWA that provides high availability, fault tolerance, and security guarantees. Thereby, an AID's witness pool constitutes a highly available and secure promulgation network for that AID.

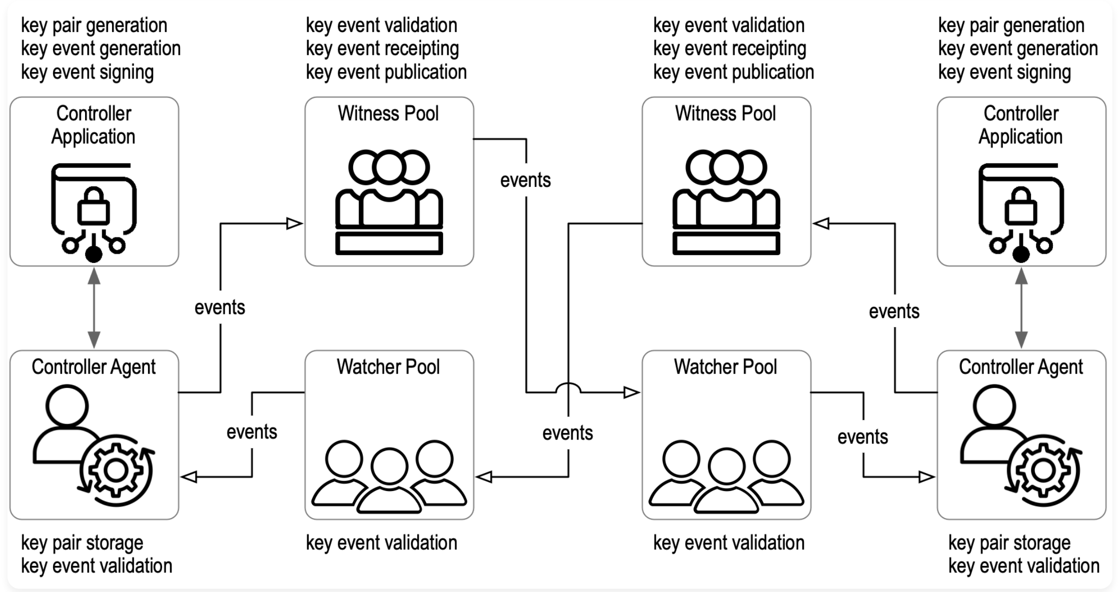
Likewise, each controller acting as a validator of some other controller's events MAY create or choose a set or pool of watchers. The validator chooses how the watchers are hosted. It MAY use a watcher service provided by some other party or MAY directly host its own watchers in its own infrastructure or some combination of the two. Nonetheless, the pool is under the ultimate control of the AID's event validator. To clarify, it is not under the control of the AID's controller. This means the validator MAY change its watcher infrastructure at will. Watchers are not AID-specific; instead, they watch the KELs of any or all AIDs that are shared with them. Watchers are not explicitly managed by key events. This is so that the watcher infrastructure used by any validator MAY be kept confidential and, therefore, unknown to potential attackers. It is up to each validator to manage its watcher infrastructure as it sees fit. Each validator uses its own watcher pool to watch the KELs of other controllers. When an AID has witnesses, the watchers of one validator watch the witnesses of the AID of some other controller. Each validator MAY use its own watcher pool to watch its own witness pool of the AID that it itself controls in order to detect external attacks on its witnesses.

Watchers MAY also exchange signed receipts of key events in the KELs they watch. Based on those receipts, a watcher pool could also employ the KAWA agreement algorithm to provide high availability, fault tolerance, and security guarantees. Thereby, a given validator's watcher pool constitutes a highly available and secure confirmation network for any AIDs from other controllers it chooses to watch.

Watchers have a strong incentive to share all the KELs they watch. This is because Watchers follow a "first seen" policy (described in more detail below). Simply put, "first seen" means that only the first Version of an event that a watcher receives is deemed by the watcher to be the one and only true version of the event. Any other versions received later are deemed invalid by that watcher, i.e., "first seen, always seen, never unseen." Thus, any later compromise of the authoritative key state for the associated AID cannot produce an alternate version of the event that could supplant the First-seen version for a given watcher. Therefore, it is in the best interests of every honest AID controller to have its original version be accepted as first-seen as widely and as quickly as possible in order to nullify any future potential exploits against its key state. It is also in the best interest of every validator to have their own watchers "first see" the earliest version of all key events from all other controllers because those earliest events are, by design, the least likely to have been compromised. This strongly incentivizes both parties to support a widespread low-latency global network of watchers and watcher pools that share their first-seen KELs.

Watchers MAY implement different additional features. A watcher could choose to keep around any verifiable key events that differ from their first-seen version. These variants are, by nature, duplicitous; in order to be verifiable, a duplicitous variant MUST be properly fully signed and witnessed. The only way that two versions of a given event can be fully signed and witnessed is if the keys of the event's controller have been compromised by an attacker or the controller itself acted duplicitously. The two cases could be indistinguishable to a watcher. However, both exhibit provable Duplicity with regard to the key state. A watcher who records and provides such evidence of duplicity to other watchers is called a Juror. A Juror MAY be a member of a highly available, fault-tolerant pool of Jurors, called a Jury. A watcher who evaluates key events based on the evidence of duplicity or lack thereof as provided by one or more Juries is called a Judge. KERI thus enables the duplicity-evident exchange of data.

Ultimately, a validator decides whether or not to trust the key state of a given AID based on the evidence or lack thereof of duplicity. A given validator MAY choose to use Judge and Jury services to aid it in deciding whether or not to trust the key state of a given AID. An honest validator MUST trust when there is no evidence of duplicity and MUST NOT trust when there is any evidence of duplicity unless and until the duplicity has been reconciled. KERI provides mechanisms for duplicity reconciliation. These include key compromise recovery mechanisms.

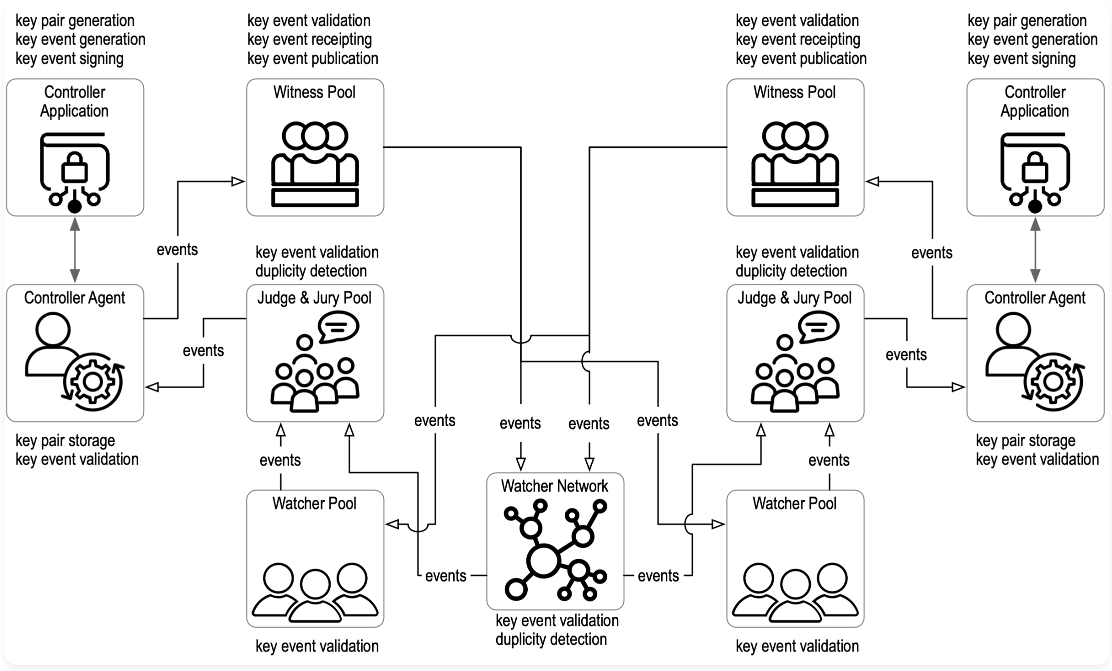


Indirect Exchange

Figure: *Indirect Exchange*

Ecosystem

The KERI protocol fosters an open, competitive ecosystem of service providers for the various infrastructure components such as controller applications (wallets, key chains, and agents), witnesses, and watchers (judges and juries). Because there is no requirement for shared governance over any of the infrastructure components, each controller and each validator is free to choose its own service providers based on price, performance, usability, etc. This enables competition across the full spectrum of infrastructure components. Thus, existing cloud and web infrastructure can be leveraged at comparable performance and price levels. KERI fosters the development of a global watcher network that will eventually result in universal duplicity detectability and ambient verifiability with the goal of providing a universal DKMI in support of a trust-spanning layer for the internet.



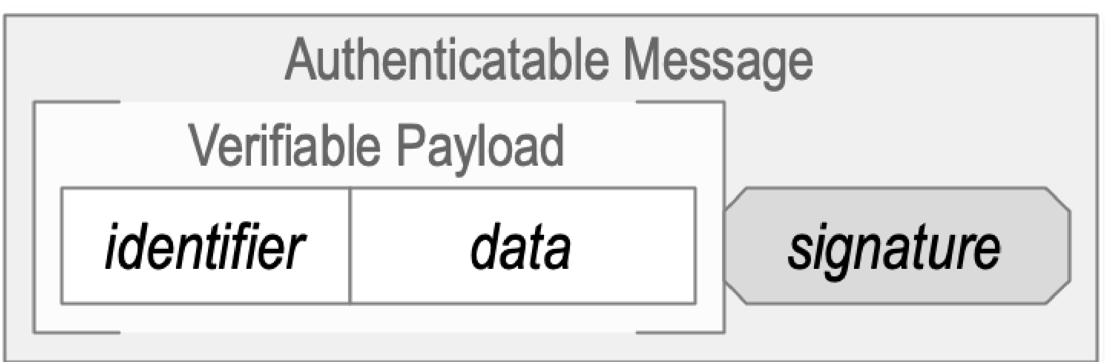
KERI Ecosystem

Figure: *KERI Ecosystem*

KERI's identifier system security overlay

The function of KERI's identifier-system security overlay is to establish the authenticity of the message payload in an IP Packet by verifiably attributing it to a Cryptonymous SCID (an AID) via an attached set of one or more asymmetric keypair-based nonrepudiable digital signatures. The current valid set of associated asymmetric keypair(s) is proven via a Verifiable data structure called the KEL. The identifier system provides a mapping between the identifier and the keypair(s) that control the identifier, namely, the public key(s) from those keypairs. The private key(s) is secret and is not shared.

An authenticatable (Verifiable) internet message (packet) or data item includes the identifier and data in its payload. Attached to the payload is a digital signature(s) made with the private key(s) from the controlling keypair(s). Given the identifier in a Message, any Verifier of a Message (data item) can use the identifier system mapping to look up the public key(s) belonging to the controlling keypair(s). The Verifier can then verify the attached signature(s) using that public key(s). Because the payload includes the identifier, the signature makes a nonrepudiable cryptographic commitment to both the source identifier and the data in the payload.



Authenticatable Message

Figure: *Authenticatable Message*

Overcoming existing security overlay flaws

KERI overcomes two major system security overlay flaws.

The first major flaw is that the mapping between the identifier (domain name) and the controlling keypair(s) is merely asserted by a trusted entity (a certificate authority or CA) via a certificate. Because the mapping is merely asserted, a Verifier cannot verify cryptographically the mapping between the identifier and the controlling keypair(s) but must trust the operational processes of the CA who issued and signed the certificate. As is well known, a successful attack upon those operational processes could fool a Verifier into trusting an invalid mapping — the certificate is issued to the wrong keypair(s) albeit with a Verifiable signature from a valid CA. Noteworthy is that the signature on the certificate is not made with the controlling keypairs of the identifier but made with keypairs controlled by the CA. The fact that the certificate is signed by the CA means that the mapping itself is not Verifiable but merely that the CA asserted the mapping between keypair(s) and identifier. The certificate merely provides evidence of the authenticity of the assignment of the mapping but not evidence of the veracity of the mapping.

The second major flaw is that when rotating the valid signing keys, there is no cryptographically Verifiable way to link the new (rotated in) controlling/signing key(s) to the prior (rotated out) controlling/signing key(s). Key rotation is asserted merely and implicitly by a

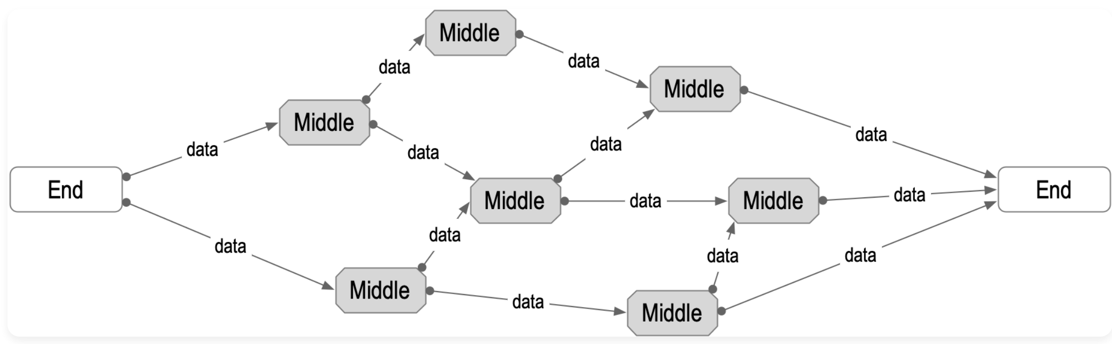
trusted entity (CA) by issuing a new certificate with new controlling/signing keys. Key rotation is necessary because over time the controlling keypair(s) of an identifier becomes weak due to exposure when used to sign Messages and must be replaced. An explicit Rotation mechanism first revokes the old keys and then replaces them with new keys. Even a certificate revocation list (CRL) [20] as per [RFC5280], with an online status protocol (OCSP) registration as per [RFC6960], does not provide a cryptographically Verifiable connection between the old and new keys; This merely is asserted. The lack of a single universal CRL or registry means that multiple potential replacements could be valid. From a cryptographic verifiability perspective, Rotation by assertion with a new certificate that either implicitly or explicitly provides revocation and replacement is essentially the same as starting over by creating a brand-new independent mapping between a given identifier and the controlling keypair(s). This start-over style of Key rotation could well be one of the main reasons that other key assignment methods, such as Pretty Good Privacy (PGP's) web-of-trust failed. Without a universally Verifiable revocation mechanism, any Rotation (revocation and replacement) assertion by some certificate authority, either explicit or implicit, is mutually independent of any other. This lack of universal cryptographic verifiability of a Rotation fosters ambiguity as to the actual valid mapping at any point in time between the identifier and its controlling keypair(s). In other words, for a given identifier, any or all assertions made by some set of CAs could be potentially valid.

The KERI protocol fixes both of these flaws using a combination of AIDs, key pre-rotation, and a Verifiable data structure, the KEL, as verifiable proof of Key state and duplicity-evident mechanisms for evaluating and reconciling Key state by Validators. Unlike certificate transparency, KERI enables the detection of Duplicity in the Key state via nonrepudiable cryptographic proofs of Duplicity, not merely the detection of inconsistency in the Key state that MAY or MAY NOT be duplicitous.

End-verifiable

A data item or statement is end-to-end-verifiable, or end-verifiable for short, when that data item could be cryptographically securely attributable to its source (party at the source end) by any recipient verifier (party at the destination end) without reliance on any infrastructure not under the verifier's ultimate control. KERI's End-verifiability is pervasive. It means that everything in KERI or that depends on KERI is also end-verifiable; therefore, KERI has no security dependency on any other infrastructure, including conventional PKI. It also does not rely on security guarantees that may or may not be provided by web or internet infrastructure. KERI's identifier system-based security overlay for the Internet provides each identifier with a primary root-of-trust based on self-certifying, self-administering, self-governing AID s and AN s that provides the trust basis for a universal AIS [4] [16] [18] [19] [17]. This root-of-trust is cryptographic, i.e. not administrative, because it does not rely on any trusted third-party administrative process but is established with cryptographically verifiable data structures alone.

Often, the two ends cannot transmit data directly between each other but relay that data through other components or infrastructure not under the control of either end. For example, Internet infrastructure is public and is not controlled by either end of a transmission. A term for any set of components that relays data between the ends or, equivalently, the party that controls it is the middle. The following diagram shows two ends communicating over the middle.



End-to-end Verifiability

Figure: *End-to-end Verifiability*

End verifiability means that the end destination can verify the source of the data without having to trust the middle. This gives rise to the concept called ambient verifiability, where the source of any data can be verified anywhere, at any time, by anybody. Ambient verifiability removes any need to trust any of the components in the middle, i.e., the whole internet.

Another term for a party at one end of a transmission over a network (internet) is a network edge. End-verifiability of data means that if the edges of the network are secure, then the security of the middle does not matter. With KERI, the security of the edges is based primarily on the security of the key management at the edges. Therefore, a KERI-based system benefits greatly because protecting one's private keys is much easier than protecting all internet infrastructure.

True end-verifiability means that only nonrepudiable digital signatures based on asymmetric, i.e., public key cryptography, can be used to securely attribute data to a cryptographically derived source identifier, i.e., AID. In this sense, data is authentic with respect to its source identifier when it is verifiably nonrepudiably signed with the authoritative keypairs for that identifier at the time of signing. The result is that KERI takes a signed-everything approach to data both in motion and at rest. This enables a no-shared-secret approach to primary authentication, i.e., the primary authenticity of a data item is not reliant on the sharing of secrets between the source and recipient of any data item. Common types of shared secrets used for authentication include passwords, bearer to-

kens, and shared encryption keys, which are all vulnerable to exploitation. The result of a no-shared-secret sign-everything approach is the strongest possible authenticity, which means secure attribution to an AID.

End verifiability implies that the infrastructure needed to verify MUST be under the ultimate control of the verifier. Otherwise, the verifier must trust in infrastructure it does not control and, therefore, can't fully verify. Thus end-verifiability is a prerequisite for true zero-trust computing infrastructure, where zero-trust means never trust always verify. The infrastructure in KERI is therefore split into two parts, the infrastructure controlled by the securely attributable source of any information where that source is identified by its AID. The source is, therefore, the AID controller. The AID controller's infrastructure is its promulgation infrastructure. The output of the promulgation infrastructure is duplicity evident in cryptographically verifiable data structures that the verifier MAY verify. The verifier MAY employ its own infrastructure to aid it in both performing cryptographic verification and detecting duplicity. The verifier's infrastructure is its confirmation infrastructure. This bifurcated architecture over verifiable data is more succinctly characterized as shared data but no shared governance. This naturally supports a no-shared-secret approach to authentication. Shared governance also usually comes with security, portability, cost, and performance limitations, which limit its more universal adoptability.

Self-certifying identifier (SCID)

The KERI identifier system overlay leverages the properties of cryptonymous SCIDs which are based on asymmetric PKI to provide end-verifiable secure attribution of any message or data item without needing to trust in any intermediary. A SCID is uniquely cryptographically derived from the public key of an asymmetric keypair, (public, private). The identifier is self-certifying in the sense that it does not rely on a trusted entity. Any nonrepudiable signature made with the private key could be verified by extracting the public key from either the identifier itself or incepting information uniquely associated with the cryptographic derivation process for the identifier. In a basic SCID, the mapping between an identifier and its controlling public key is self-contained in the identifier itself. A basic SCID is ephemeral i.e., it does not support Rotation of its keypairs in the event of key weakness or compromise and therefore MUST be abandoned once the controlling private key becomes weakened or compromised from exposure. The class of identifiers that generalize SCIDs with enhanced properties such as persistence is called AIDs.

Autonomic identifier (AID)

The use of a KEL gives rise to an enhanced class of SCIDs that are persistent, i.e. not ephemeral, because the SCID's keys could be refreshed or updated via Rotation, allowing secure control over the identifier in spite of key weakness or even compromise. Members of this family of generalized enhanced SCIDs are called AIDs. Autonomic means self-governing, self-regulating, or self-managing and is evocative of the self-certifying, self-managing, and self-administering properties of this class of identifier. An AID

could exhibit other self-managing properties, such as transferable control using key pre-rotation, which enables control over such an AID to persist in spite of key weakness or compromise due to exposure. Authoritative control over the identifier persists in spite of the evolution of the Key state.

Key rotation/pre-rotation

An important innovation of KERI is that it solves the key Rotation problem of PKI (including that of simple SCIDs) via a novel but elegant mechanism called key pre-rotation. This pre-rotation mechanism enables an entity to persistently maintain or regain control over an identifier in spite of the exposure-related weakening over time or even compromise of the current set of controlling (signing) keypairs. With key pre-rotation, control over the identifier can be re-established by rotating to a one-time use set of unexposed but pre-committed rotation keypairs that then become the current signing keypairs. Each Rotation, in turn, cryptographically commits to a new set of rotation keys but without exposing them. Because the pre-rotated keypairs need never be exposed prior to their one-time use, their attack surface could be optimally minimized. The current Key state is maintained via a KEL, an append-only Verifiable data structure. Cryptographic verifiability of the Key state over time is essential to remove this ambiguity over the mapping between the identifier (domain name) and the controlling keypair(s). Without this verifiability, the detection of potential ambiguity requires yet another bolt-on security overlay, such as the Certificate transparency system.

Qualified Cryptographic Primitives

A Cryptographic primitive is a serialization of a value associated with a cryptographic operation, including but not limited to a digest (hash), a Salt, a seed, a private key, a public key, or a signature. Furthermore, a Qualified cryptographic Primitive includes a prepended derivation code (as a proem) that indicates the cryptographic algorithm or suite used for that derivation; e.g. `EL1L56LyoKrIofnn0oPChS4EyzMHEEk75INJohDS_Bug`. This simplifies and compactifies the essential information needed to use that Cryptographic primitive. All Cryptographic primitives in KERI MUST be expressed using the CESR (Compact Event Streaming Representation) protocol [1]. A property of CESR is that all cryptographic primitives expressed in either its Text or Binary domains are qualified by construction. Indeed, cryptographic primitive qualification is an essential property of CESR which makes a uniquely beneficial encoding for a cryptographic primitive heavy protocol like KERI.

CESR Encoding

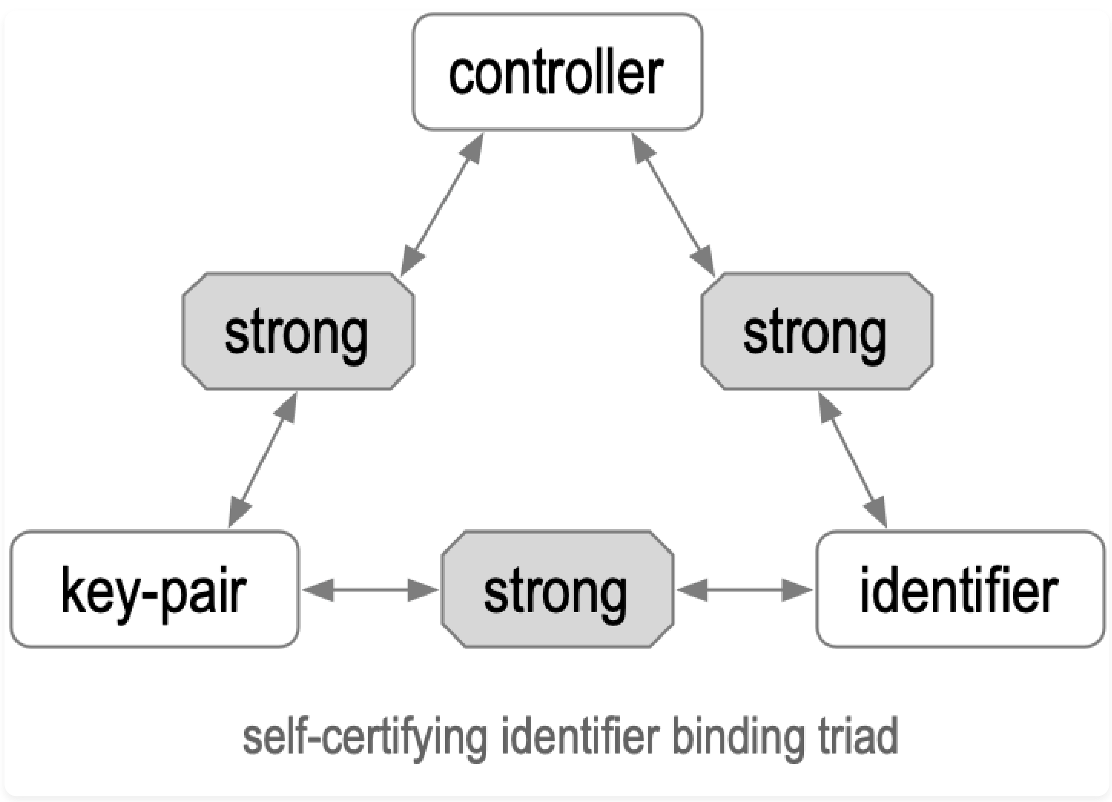
As stated previously, KERI represents all cryptographic primitives with [1]. CESR supports round-trip lossless conversion between its Text, Binary, and Raw domain representations and lossless composability between its Text and Binary domain representations. Composability is ensured between any concatenated group of text Primitives and the bi-

nary equivalent of that group because all CESR Primitives are aligned on 24-bit boundaries. Both the text and binary domain representations are serializations suitable for transmission over the wire. The Text domain representation is also suitable to be embedded as a field or array element string value as part of a field map serialization such as JSON, CBOR, or MsgPack. The Text domain uses the set of characters from the URL-safe variant of Base64, which in turn is a subset of the ASCII character set. For the sake of readability, all examples in this specification are expressed in CESR's Text domain.

The CESR protocol supports several different types of encoding tables for different types of derivation codes used to qualify primitives. These tables include very compact codes. For example, a 256-bit (32-byte) digest using the BLAKE3 digest algorithm, i.e., Blake3-256, when expressed in Text domain CESR, consists of 44 Base64 characters that begin with the one-character derivation code `E`, such as `EL1L56LyoKrIofnn0oPChS4EyzMHEEk75INJohDS_Bug`. The equivalent qualified Binary domain representation consists of 33 bytes. Unless otherwise indicated, all Cryptographic primitives used in this specification are qualified Primitives expressed in CESR's Text domain. This includes serializations that are signed, hashed, or encrypted.

KERI's secure bindings

In simple form, an identifier-system security overlay binds together a triad consisting of the identifier, keypairs, and Controllers, the set of entities whose members control a private key from the given set of keypairs. The set of Controllers is bound to the set of keypairs, the set of keypairs is bound to the identifier, and the identifier is bound to the set of Controllers. This binding triad can be diagrammed as a triangle where the sides are the bindings and the vertices are the identifier, the set of Controllers, and the set of key pairs. This triad provides verifiable control authority for the identifier.

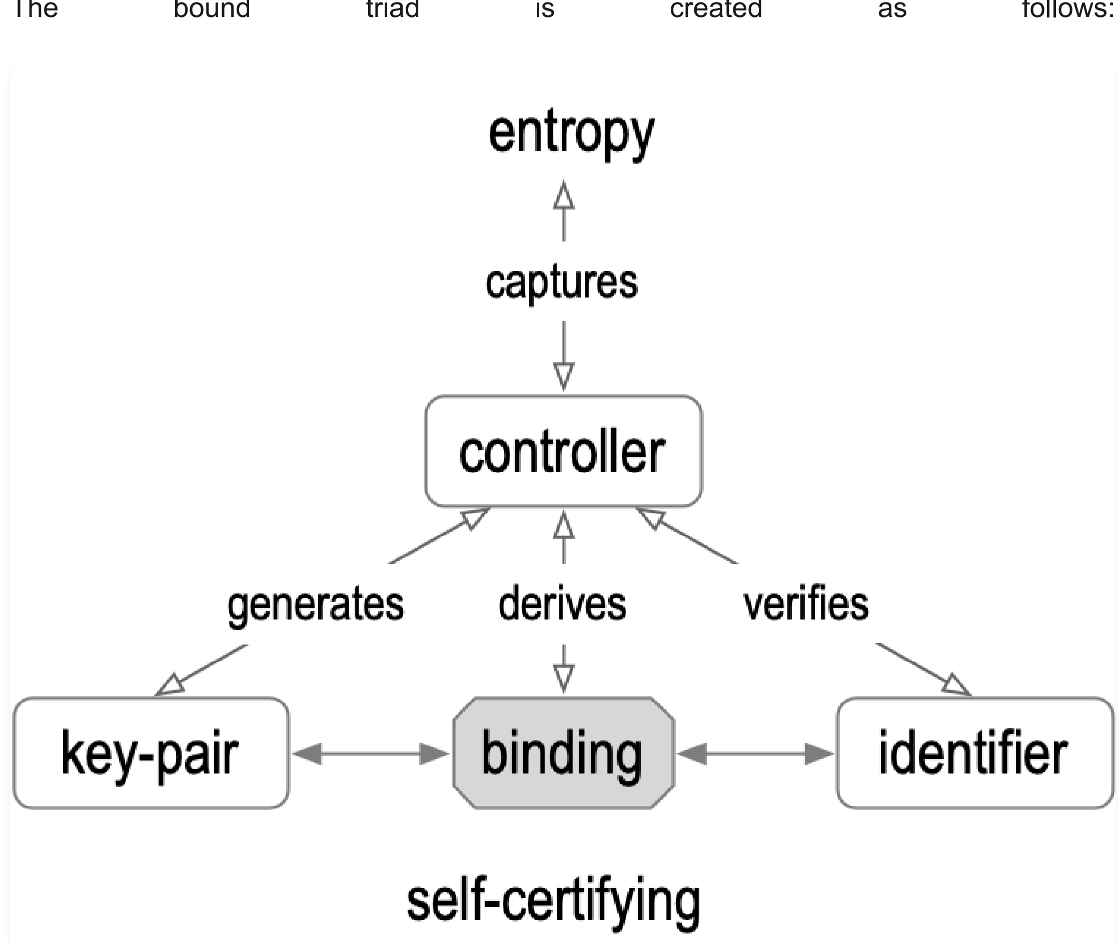


Self-certifying Identifier Binding Triad

Figure: *Self-certifying Identifier Binding Triad*

When these bindings are strong, then the overlay is highly invulnerable to attack. In contrast, when these bindings are weak, then the overlay is highly vulnerable to attack. With KERI, all the bindings of the triad are strong because they are cryptographically Verifiable with a minimum cryptographic strength or level of approximately 128 bits. See Annex A on cryptographic strength for more detail.

The bound triad is created as follows:



Self-certifying Issuance Triad

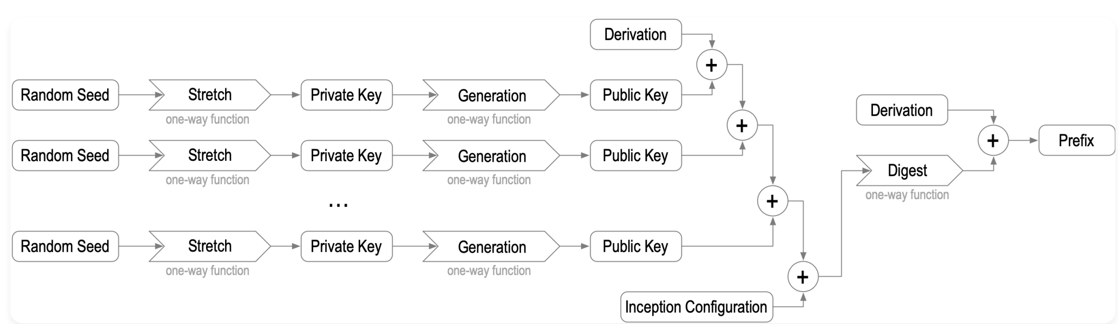
Figure: *Self-certifying Issuance Triad*

Each Controller in the set of Controllers creates an asymmetric (public, private) keypair. The public key is derived from the private key or seed using a one-way derivation that MUST have a minimum cryptographic strength of approximately 128 bits (nominally). Depending on the crypto-suite used to derive a keypair, the private key or seed could itself have a length larger than 128 bits. A Controller could use a cryptographic strength pseudo-random number generator (CSPRNG) to create the private key or seed material.

Because the private key material MUST be kept secret (by definition), typically in a secure data store, the management of those secrets could be an important consideration. One approach to minimize the size of secrets is to create private keys or seeds from a secret salt. The salt must have an entropy of approximately 128 bits. Then, the salt could be stretched to meet the length requirements for the crypto suite's private key size. In addition, a hierarchical deterministic derivation function could be used to further minimize storage requirements by leveraging a single salt for a set or sequence of private keys.

Because each Controller is the only entity in control (custody) of the private key, and the public key is universally uniquely derived from the private key using a cryptographic strength one-way function, then the binding between each Controller and their keypair is as strong as the ability of the Controller to keep that key private. The degree of protection is up to each Controller to determine. For example, a Controller could choose to store their private key in a safe at the bottom of a coal mine, air-gapped from any network, with an ex-special forces team of guards. Or the Controller could choose to store it in an encrypted data store (key chain) on a secure boot mobile device with a biometric lock or simply write it on a piece of paper and store it in a safe place. The important point is that the strength of the binding between the Controller and keypair does not need to be dependent on any trusted entity.

The identifier is derived universally and uniquely from the set of public keys using a one-way derivation function. It is, therefore, an AID (qualified SCID). Associated with each identifier (AID) is incepting information that MUST include a list of the set of qualified public keys from the controlling keypairs. In the usual case, the identifier is a qualified cryptographic digest of the serialization of all the incepting information for the identifier. Any change to even one bit of the incepting information changes the digest and hence changes the derived identifier. This includes any change to any one of the qualified public keys, including its qualifying derivation code. To clarify, a qualified digest as an identifier includes a derivation code as a proem that indicates the cryptographic algorithm used for the digest. Thus, a different digest algorithm results in a different identifier. In this usual case, the identifier is bound strongly and cryptographically to the public keys and any other incepting information from which the digest was generated.



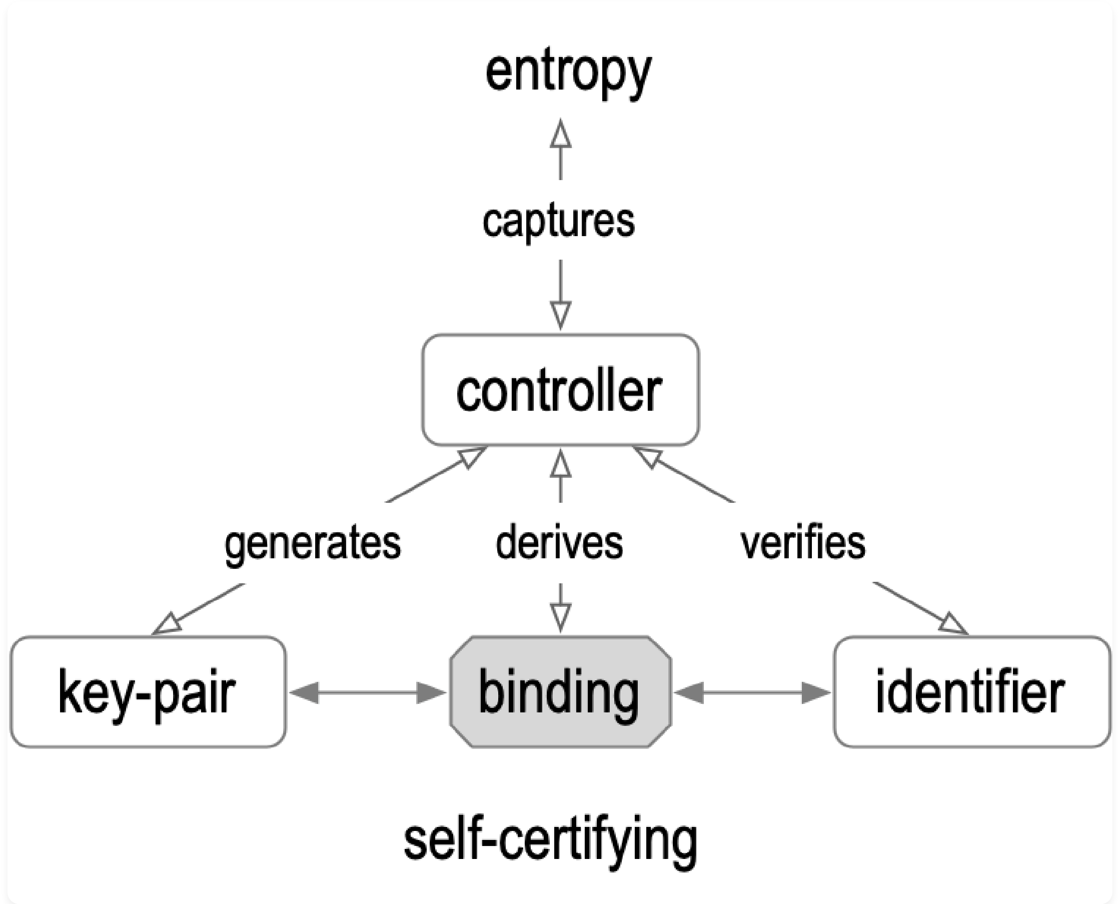
AID Identifier Prefix Derivation

Figure: *AID Identifier Prefix Derivation*

A special case could arise when the set of public keys has only one member, i.e., there is only one controlling keypair. In this case, the Controller of the identifier could choose to use only the qualified public key as the identifier instead of a qualified digest of the incepting information. In this case, the identifier is still strongly bound to the public key but not to any other incepting information. A variant of this single keypair special case is an identifier that cannot be rotated. Another way of describing an identifier that cannot be ro-

tated is that it is a non-transferable identifier because control over the identifier cannot be transferred to a different set of controlling keypairs. In contrast, a rotatable keypair is transferable because control can be transferred via rotation to a new set of keypairs. Essentially, when non-transferable, the identifier's lifespan is ephemeral, not persistent, because any weakening or compromise of the controlling keypair means that the identifier MUST be abandoned. Nonetheless, there are important use cases for an ephemeral AID. In all cases, the derivation code in the identifier indicates the type of identifier, whether it be a digest of the incepting information (multiple or single keypair) or a single member special case derived from only the public key (both ephemeral or persistent).

Each Controller in a set of Controllers can prove its contribution to the control authority over the identifier in either an interactive or non-interactive fashion. One form of interactive proof is to satisfy a challenge of that control. The challenger creates a unique challenge Message. The Controller responds by nonrepudiably signing that challenge with the private key from the keypair under its control. The challenger can then cryptographically verify the signature using the public key from the Controller's keypair. One form of non-interactive proof is the periodic contribution to a monotonically increasing sequence of nonrepudiably signed updates of some data item. Each update includes a monotonically increasing sequence number or date-time stamp. Any Verifier then can cryptographically verify the signature using the public key from the Controller's keypair and verify that the update was made by the Controller. In general, only members of the set of Controllers can create verifiable, nonrepudiable signatures using their keypairs. Consequently, the identifier is strongly bound to the set of Controllers via provable control over the keypairs.



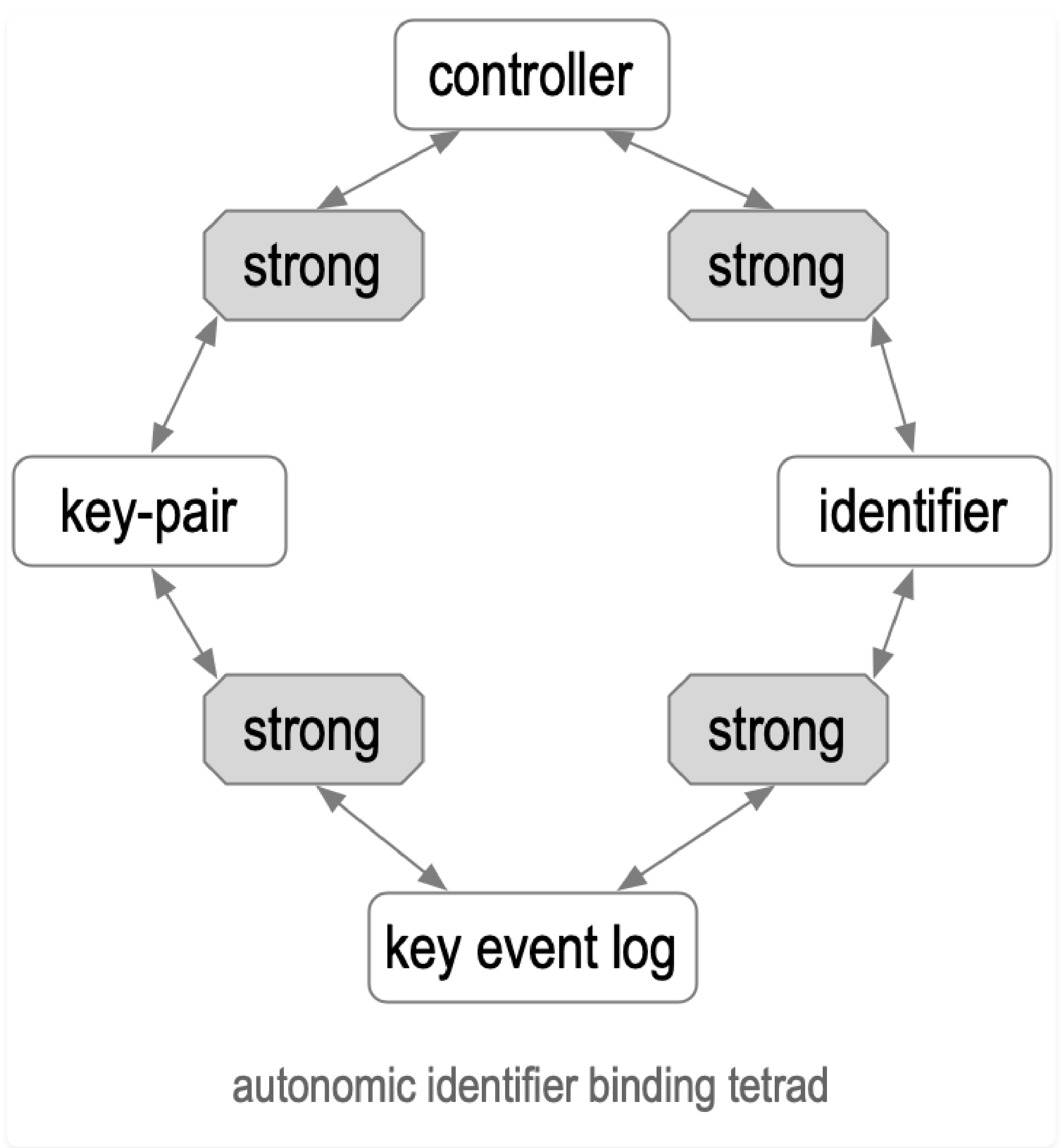
Self-certifying Identifier Issuance Triad

Figure: *Self-certifying Identifier Issuance Triad*

Tetrad bindings

At Inception, the triad of an identifier, a set of keypairs, and a set of Controllers are strongly bound together. But in order for those bindings to persist after a key Rotation, another mechanism is required. That mechanism is the KEL, a Verifiable data structure [4] [21]. The KEL is not necessary for non-transferable identifiers that do not need to persist control via key Rotation despite key weakness or compromise. To reiterate, transferable (persistent) identifiers each need a KEL; non-transferable (ephemeral) identifiers do not.

For persistent (transferable) identifiers, this additional mechanism can be bound to the triad to form a tetrad consisting of the KEL, the identifier, the set of keypairs, and the set of Controllers. The first entry in the KEL is called the Inception event, a serialization of the incepting information associated with the previously mentioned identifier.



Autonomic Identifier Binding Tetrad

Figure: *Autonomic Identifier Binding Tetrad*

The Inception event MUST include the list of controlling public keys and a signature threshold and be signed by a set of private keys from the controlling keypairs that satisfy that threshold. Additionally, for transferability (persistence across Rotation), the Inception event MUST include a list of digests of the set of pre-rotated public keys and a pre-rotated signature threshold that will become the controlling (signing) set of key keypairs and threshold after a Rotation. A non-transferable identifier MAY have a trivial KEL that only includes an Inception event but with a null set (empty list) of pre-rotated public keys.

A Rotation is performed by appending a Rotation event to the KEL. A Rotation event MUST include a list of the set of pre-rotated public keys (not their digests), thereby exposing them, and be signed by a set of private keys from these newly exposed newly

controlling but pre-rotated keypairs that satisfy the pre-rotated threshold. The Rotation event MUST include a list of the digests of a new set of pre-rotated keys as well as the signature threshold for the set of pre-rotated keypairs. At any point in time, the transferability of an identifier MAY be removed via a Rotation event that rotates to a null set (empty list) of pre-rotated public keys. Rotating to a null set of pre-rotated public keys signals to any verifier that the controller has abandoned the identifier.

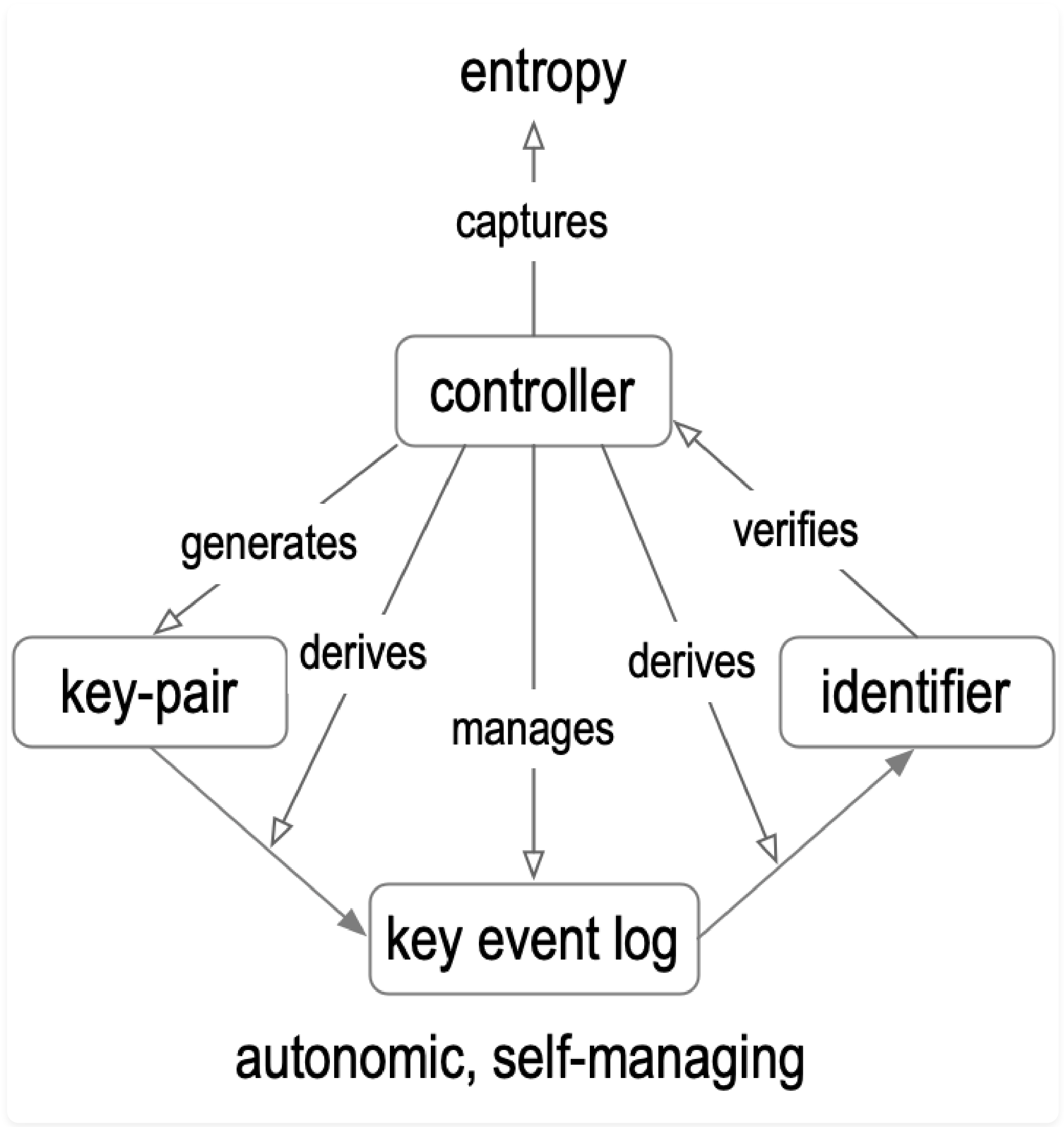
Each event in a KEL MUST include an integer sequence number that is one greater than the previous event. The key event log is meant to be duplicity evident. The sn makes this easier to detect. Each event after the Inception event also MUST include a cryptographic digest of the previous event. This digest means that a given event is bound cryptographically to the previous event in the sequence. The list of digests or pre-rotated keys in the Inception event cryptographically binds the Inception event to a subsequent Rotation event, essentially making a forward commitment that forward chains together the events. The only valid Rotation event that MAY follow the Inception event MUST include the pre-rotated keys. But only the Controller who created those keys and created the digests can verifiably expose them. Each Rotation event, in turn, makes a forward commitment (chain) to the following Rotation event via its list of pre-rotated key digests. This makes the KEL a doubly (backward and forward) hash (digest) chained nonrepudiably signed append-only Verifiable data structure.

Because the signatures on each event are nonrepudiable, the existence of an alternate but Verifiable KEL for an identifier is provable evidence of Duplicity. In KERI, there MUST be at most one valid KEL for any identifier or none at all. Any Validator of a KEL can enforce this one valid KEL rule that protects the Validator before relying on the KEL as proof of the current key state for the identifier. Any irreconcilable evidence of Duplicity means the Validator does not trust (rely on) any KEL to provide the key state for the identifier. Rules for handling reconcilable Duplicity will be discussed below in section Reconciliation. From a Validator's perspective, either there is one-and-only-one valid KEL or none at all, which also protects the Validator by removing any potential ambiguity about the Key state. The combination of a Verifiable KEL made from nonrepudiably signed backward and forward hash chained events together with the only-one-valid KEL rule strongly binds the identifier to its current Key state as given by that one valid KEL (or not at all). This, in turn, binds the identifier to the Controllers of the current keypairs given by the KEL, thus completing the tetrad.

At Inception, the KEL can be bound even more strongly to its tetrad by deriving the identifier from a digest of the Inception event so that even one change in any of the incepting information included in the Inception event will result in a different identifier (including not only the original controlling keys pairs but also the pre-rotated keypairs).

The essence of the KERI protocol is a strongly bound tetrad of an identifier, set of keypairs, set if Controllers, and the KEL that forms the basis of its identifier system security overlay. The KERI protocol introduces the concept of Duplicity evident programming via

Duplicity evident Verifiable data structures.



Autonomic Identifier Issuance Tetrad

Figure: *Autonomic Identifier Issuance Tetrad*

Autonomic Namespaces (ANs)

A namespace groups symbols or identifiers for a set of related objects [23]. In an identity system, an identifier can be generalized as belonging to a namespace that provides a systematic way of organizing related identifiers with their resources and attributes.

To elaborate, a namespace employs some scheme for assigning identifiers to the elements of the namespace. A simple name-spacing scheme uses a prefix or prefixes in a hierarchical fashion to compose identifiers. The following is an example of a namespace scheme for addresses within the USA that uses a hierarchy of prefixes:

```
state.county.city.zip.street.number.
```

An example element in this namespace could be identified with the following:

```
utah.wasatch.heber.84032.main.150S.
```

where each prefix location has been replaced with the actual value of the element of the address. Namespaces provide a systematic way of organizing related elements and are widely used in computing.

An Autonomic namespace , AN, is defined as a namespace with an AID as a prefix, i.e., a fully qualified CESR-encoded cryptographic primitive. As defined above, AIDs are uniquely (strongly) cryptographically bound to their incepting controlling keypair(s) at issuance. They are, hence, self-certifying. In addition, AIDs are also bound to other key management information, such as the hashes of the next pre-rotated rotation keys and their witnesses. This makes them self-managing.

To clarify, each identifier from an AN includes as a prefix an identifier encoded in CESR that either is the public key or is uniquely cryptographically derived from the public key(s) of the incepting (public, private) key pair. The controller can then use the associated private key(s) to authoritatively (nonrepudiably) sign statements that authenticate and authorize the use of the identifier. These statements include responses to challenges to prove control over the identifier. Thus, self-certification enables both self-authentication and self-authorization capabilities as well as self-management of cryptographic signing keypairs. Together, these properties make the namespace self-administering.

To restate, An AN is self-administrating. The self-administering properties of an AN mean that the resources identified by that namespace can also be governed by the controller of the key pair(s) that are authoritative for its AID prefix. Essentially, a namespace of identifiers can be imbued with the secure attributability properties of an AID when used as its prefix. When secure attribution arises solely from the AID as the prefix, then the other elements in the namespace syntax are immaterial with respect to secure attribution. This makes the prefix agnostic about the syntax of any ANs in which it appears. The same AID can be used to control multiple disparate namespaces. This provides opportunities for AIDs to interoperate with many existing namespaces. The AID prefix from any autonomically namespaced identifier merely needs to be extracted to use the prefix with a KERI protocol library.

In general, a fully qualified AID primitive and an identifier from an AN based on that primitive as a prefix can both be referred to as AIDs with the understanding that any cryptographic operations only apply to the prefix portion of the namespaced identifier.

The primary purpose of an AID is to enable any entity to establish control over its associated identifier namespace in an independent, interoperable, and portable way. This approach builds on the idea of an identity (identifier) meta-system that enables interoperability between systems of identity (identifiers) that not only exposes a unified interface but adds decentralized control over their identifiers. This enables portability, not just interoperability. Given portability in an identity (identifier) meta-system, transitive trust can occur, that is, the transfer of trust between contexts or domains. Because transitive trust facilitates the transfer of other types of value, a portable decentralized identity meta-system, e.g., an AID system, enables an identity meta-platform for commerce.

KERI data structures and labels

KERI data structure format

A KERI data structure, such as a Key event message body, can be abstractly modeled as a nested key:value mapping. To avoid confusion with the cryptographic use of the term key, the term field is used instead herein to refer to a mapping pair, with the terms field label and field value used to refer to each pair member. Two tuples can represent these pairs, e.g., (label, value). When necessary, this terminology is qualified by using the term field map to reference such a mapping. Field maps can be nested where a given field value is itself a reference to another field map and are referred to as a nested field map or simply a nested map for short.

A field can be represented by a framing code or a block-delimited serialization. In a block delimited serialization, such as JSON, each field map is represented by an object block with block delimiters such as `{}`. Given this equivalence, the term block or nested block can be used as synonymous with field map or nested field map. In many programming languages, a field map is implemented as a dictionary or hash table. This enables performant asynchronous lookup of a field value from its field label. Reproducible serialization of field maps requires a canonical ordering of those fields. One such canonical ordering is called insertion or field creation order. A list of (field, value) pairs provides an ordered representation of any field map.

Most programming languages now support ordered dictionaries or ordered hash tables that provide reproducible iteration over a list of ordered field (label, value) pairs where the ordering is the insertion or field creation order. This enables reproducible round-trip serialization/deserialization of field maps. Serialized KERI data structures depend on insertion-ordered field maps for their canonical serialization/deserialization. KERI data structures support multiple serialization types, namely JSON, CBOR, MGPK, and CESR but for the sake of simplicity, JSON only will be used for examples. The basic set of normative field labels in KERI field maps is defined by the table below (see the next section).

KERI field labels for data structures

Reserved field labels in Keri messages:

Label	Title	Description
`v`	Version String	enables regex parsing of field map in CESR stream
`t`	Message Type	three character string
`d`	Digest SAID	fully qualified digest of block in which it appears
`i`	Identifier Prefix (AID)	fully qualified primitive, Controller AID
`s`	Sequence Number	strictly monotonically increasing integer encoded in hex
`p`	Prior SAID	fully qualified digest, prior message SAID
`kt`	Keys Signing Threshold	hex encoded integer or fractional weight list
`k`	List of Signing Keys (ordered key set)	list of fully qualified primitives
`nt`	Next Keys Signing Threshold	hex encoded integer or fractional weight list
`n`	List of Next Key Digests (ordered key digest set)	list of fully qualified primitives digests
`bt`	Backer Threshold	hex encoded integer
`b`	List of Backers (ordered backer set of AIDs)	list of fully qualified primitives
`br`	List of Backers to Remove (ordered backer set of AIDS)	list of fully qualified primitives
`ba`	List of Backers to Add (ordered backer set of AIDs)	list of fully qualified primitives
`c`	List of Configuration Traits/Modes	list of strings

Label	Title	Description
`a`	List of Anchors (seals)	list of field maps
`di`	Delegator Identifier Prefix (AID)	fully qualified primitive, Delegator AID

A field label MAY have different values in different contexts but MUST NOT have a different field value type. This REQUIREMENT makes it easier to implement in strongly typed languages with rigid data structures. Notwithstanding the former, some field value types MAY be a union of elemental value types.

Because the order of field appearance MUST be enforced in all KERI data structures, whenever a field appears (in a given Message or block in a Message), the message in which a label appears MUST provide the necessary context to determine the meaning of that field fully and hence the field value type and associated semantics.

Compact KERI field labels

The primary field labels are compact in that they use only one or two characters. KERI is meant to support resource-constrained applications such as supply chain or IoT (Internet of Things) applications. Compact labels better support resource-constrained applications in general. With compact labels, the over-the-wire verifiable signed serialization consumes a minimum amount of bandwidth. Nevertheless, without loss of generality, a one-to-one normative semantic overlay using more verbose expressive field labels can be applied to the normative compact labels after verification of the over-the-wire serialization. This approach better supports bandwidth and storage constraints on transmission while not precluding any later semantic post-processing. This is a well-known design pattern for resource-constrained applications.

Special label ordering requirements

The top-level fields of each message type MUST appear in a specific order. All top-level fields are REQUIRED. This enables compact top-level fixed field CESR native messages. These top-level fields are defined for each message type below.

Version string field

The version string, `v`, field MUST be the first field in any top-level KERI field map encoded in JSON, CBOR, or MGPK as a message body [RFC4627] [2] [RFC8949] [3]. The detailed description of the version string is provided in the CESR protocol specification [1]. In summary here, it provides a regular expression target for determining a serialized field map's serialization format and size (character count) constituting an KERI message body. The Regexable format is `KERIMmmGggKKKKSSSS.` that provides protocol type `KERI`, major protocol version `M`, minor protocol version `mm`, major genus version `G`, minor genus version `gg`, serialization type `KKKK`, size `SSSS`, and terminator `.`.

To elaborate, the protocol field, `PPPP` value in the Version String MUST be `KERI` for the KERI protocol. The protocol version field, `Mmm`, MUST encode the current major `M` and minor `mm` version of the KERI protocol [1] used by the associated message. The CESR genus version field `Ggg` MUST encode the major `G` and minor `gg` version of the CESR protocol used to encode the associated message [1].

At the top-level, a stream parser can use the version string to extract and deserialize (deterministically) any serialized stream of KERI message bodies. Each KERI message body at the top-level of a stream MAY use a different serialization type. A more detailed format specification for the version string field value is found in the CESR specification [1].

Legacy version string field format

Compliant KERI version 2.XX implementations MUST support the old KERI version 1.x version string format to properly verify message bodies created with 1.x format events. The old version 1.x version string format is defined in the CESR specification [1]. The protocol field, `PPPP` value in the version string MUST be `KERI` for the KERI protocol. The version field, `vv`, MUST encode the old version of the KERI protocol [1].

Message type field

The message type, `t` field value MUST be a three-character string that provides the message type. There are three classes of message types in KERI. The first class consists of key event messages. These are part of the KEL for an AID. A subclass of key event messages are Establishment event messages, these determine the current key state. Non-establishment event messages are key event messages that do not change the key state. The second class of messages consists of Receipt messages. These are not themselves part of a KEL but convey proofs such as signatures or seals as attachments to a key event. The third class of messages consists of various message types not part of a KEL but are useful for managing the information associated with an AID.

The message types in KERI are detailed in the table below:

Type	Title	Class	Description
	Key Event Messages		
`icp`	Inception	Establishment Key Event	Incepts an AID and initializes its keystate
`rot`	Rotation	Establishment Key Event	Rotates the AID's key state
`ixn`	Interaction	Non-Establishment Key Event	Seals interaction data to the current key state
`dip`	Delegated Inception	Establishment Event	Incepts a Delegated AID and initializes its keystate
`drt`	Delegated Rotation	Establishment Key Event	Rotates the Delegated AID's key state
	Receipt Messages		
`rct`	Receipt	Receipt Message	Associates a proof such as signature or seal to a key event
	Routed Messages		
`qry`	Query	Other Message	Query information associated with an AID
`rpy`	Reply	Other Message	Reply with information associated with an AID either solicited by

Type	Title	Class	Description
			Query or unsolicited
`pro`	Prod	Other Message	Prod (request) information associated with a Seal
`bar`	Bare	Other Event	Bare (response) with information associated with a Seal either solicited by Prod or unsolicited
`xip`	Exchange Inception	Other Message	Incepts multi-exchange message transaction, the first exchange message in a transaction set
`exn`	Exchange	Other Message	Generic exchange of information, MAY be a member of a multi-message transaction set

SAID fields

Some fields in KERI data structures can have a SAID (self-referential content addressable), as a field value. In this context, `d` is short for digest, which is short for SAID. A SAID follows the SAID protocol. A SAID is a special type of cryptographic digest of its encapsulating field map (block). The encapsulating block of a SAID is called a SAD (Self-Addressed Data). Using a SAID as a field value enables a more compact but secure representation of the associated block (SAD) from which the SAID is derived. Any nested field map that includes a SAID field (i.e., is, therefore, a SAD) MAY be compacted into its SAID. The uncompacted blocks for each associated SAID MAY be attached or cached to optimize bandwidth and availability without decreasing security.

Each SAID provides a stable universal cryptographically verifiable and agile reference to its encapsulating block (serialized field map).

A cryptographic commitment (such as a digital signature or cryptographic digest) on a given digest with sufficient cryptographic strength including collision resistance is equivalent to a commitment to the block from which the given digest was derived. Specifically, a digital signature on a SAID makes a Verifiable cryptographic nonrepudiable commitment that is equivalent to a commitment on the full serialization of the associated block from which the SAID was derived. This enables reasoning about KERI data structures in whole or in part via their SAIDS in a fully interoperable, Verifiable, compact, and secure manner. This also supports the well-known bow-tie model of Ricardian Contracts [22]. This includes reasoning about the whole KERI data structure given by its top-level SAID, `d`, field as well as reasoning about any nested or attached data structures using their SAIDS.

The SAID, `d` field is the SAID of its enclosing block (field map); when it appears at the top level of the message, it is the SAID of the message itself.

The prior, `p` field is the SAID of a prior event message. When the prior `p` field appears in a key event message, then its value MUST be the SAID of the key event message whose sequence number is one less than the sequence number of its own key event message. Only key event messages have sequence numbers. Routed messages do not. When the prior, `p` field appears in an Exchange, `exn` message then its value is the SAID of the prior exchange message in the associated exchange transaction.

AID fields

Some fields, such as the `i` and `di` fields, MUST each have an AID as its value. An AID is a fully qualified primitive as described above (KERI) [4].

In this context, `i` is short for `ai`, which is short for the Autonomic identifier (AID). The AID given by the `i` field can also be thought of as a securely attributable identifier, authoritative identifier, authenticatable identifier, authorizing identifier, or authoring identifier. Another way of thinking about an `i` field is that it is the identifier of the authoritative entity to which a statement can be securely attributed, thereby making the statement verifiably authentic via a nonrepudiable signature made by that authoritative entity as the Controller of the private key(s).

The Controller AID, `i` field value is an AID that controls its associated KEL. When the Controller Identifier AID, `i` field appears at the top-level of a key event, `[icp, rot, ixn, dip, drt]` or a receipt, `rct` message it refers to the Controller of the associated KEL. When the Controller Identifier AID, `i` field appears at the top-level of an Exchange Transaction Inception, `xip` or Exchange, `exn` message it refers Controller AID of the sender of that message. A Controller AID, `i` field MAY appear in other places in mes-

sages. In those cases, its meaning SHOULD be determined by the context of its appearance. The value of the di (Delegator identifier AID) field in a Delegated Inception dip event is the AID of the Delegator.

Sequence number field

The Sequence Number, `s` field value is a hex encoded (no leading zeros) non-negative strictly monotonically increasing integer. The Sequence Number, `s` field value in Inceptions, `icp` and Delegated Inception, `dip` events MUST be `0` (hex encoded 0). The Sequence number value of all subsequent key events in a KEL MUST be 1 greater than the previous event. The maximum value of a sequence number MUST be `ffffffffffffffffffffffffffffffff` which is the hex encoding of `2^128 - 1 = 340282366920938463463374607431768211455`. This is large enough to be deemed computationally infeasible for a KEL ever to reach the maximum.

Key and key digest threshold fields

The Key Threshold, `kt` and Next Key Digest Threshold, `nt` field values each provide signing and rotation thresholds, respectively, for the key and next key digest lists. The threshold value MAY be either the simple case of a hex-encoded non-negative integer or the complex case of a list of clauses of fractional weights. The latter is called a fractionally weighted threshold.

In the simple case, given a threshold value `M` together with a total of `N` values in a key or key digest list, then the satisfaction threshold is an `M of N` threshold. This means that any set of `M` valid signatures from the keys in the list satisfies such a threshold.

In the complex case, the field value is a list of weights that are strict decimal-encoded rational fractions. Fractionally weight thresholds are best suited for Partial, Reserve, or Custodial rotation applications. The exact syntax and satisfaction properties of fractionally weighted threshold values are described below in the section on Partial, Reserve, and Custodial rotations.

Key list field

The Key, `k` field value is a list of strings that are each a fully qualified public key. These provide the current signing keys for the AID associated with a KEL. The Key, `k` field value MUST NOT be empty.

Next key digest list field

The Next Key Digest, `n` field value is a list of strings that are each a fully qualified digest of a public key. These provide the next rotation keys for the AID associated with a KEL. When the Next, `n` field value in an Inception or Delegated Inception event is an empty list, then the associated AID MUST be deemed non-transferable, and no more key

events MUST be allowed in that KEL. When the Next, `n` field value in a Rotation or Delegated Rotation Event event is an empty list, then the associated AID MUST be deemed abandoned, and no more key events MUST be allowed in its KEL.

Backer threshold field

The Backer threshold, `bt` field value is a hex-encoded non-negative integer. This is the number of backers in the backer list that MUST support a key event for it to be valid. Witness Backers express support via a signature (endorsement) of the key event that MAY be conveyed via a Receipt message. Ledger Registrar backers MAY express support by anchoring the key event or the SAID of the key event on the associated ledger.

Given a threshold value `M` together with a total of `N` values in the Backer list, then the satisfaction threshold is an `M of N` threshold. This means that any set of `M` valid endorsements (signatures or other) from the `N` Backers in the Backer list satisfies the threshold. The KAWA (Keri Algorithm of Witness Agreement) section provides a formula for determining a sufficient `M` to ensure agreement in spite of a number of `F` faulty backers.

When the Backer, `b` field value is an empty list, then the Backer Threshold, bt, field value MUST be `0` (hex-encoded 0).

Backer list

The Backer, `b` field value is a list of strings that each is the fully qualified AID of a Backer. A given AID MUST NOT appear more than once in any Backer list. These provide the current AIDs of the backers of a KEL. The list MAY be empty. When the Backers are Witnesses, then the AIDs themselves MUST be non-transferable, fully qualified public keys. As a result, a Validator can verify a witness signature given the Witness AID as public key. Consequently, the Witness does not need a KEL because its key state is fixed and is given by its AID. Although a Witness does not need a KEL, it MAY have one that consists of a trivial Inception event with an empty Next, `n` field list (making it non-transferable).

Backer remove list

The Backer Remove, `br` field value is a list of strings that each is the fully qualified AID of a Backer to be removed from the current Backer list. This allows Backer lists to be changed in an incremental fashion. A given AID MUST not appear more than once in any Backer Remove list. The Backer Remove, `br` list appears in Rotation and Delegated Rotation events. Given such an event, the current backer list is updated by removing the AIDs in the Backer Remove, `br` list. The AIDs in the Backer Remove, `br` list MUST be removed before any AIDs in the Backer Add, `ba` list are appended.

Backer add list

The Backer Add, ba field value is a list of strings, each representing the fully qualified AID of a Backer to be appended to the current Backer list. This allows Backer lists to be changed in an incremental fashion. A given AID MUST NOT appear more than once in any Backer Add list. The Backer Add, `ba` list appears in Rotation and Delegated Rotation events. Given such an event, the current backer list is updated by appending in order the AIDs from the Backer Add, `ba` list except for any AIDs that already appear in the current Backer list. The AIDs in the Backer Add, `ba` list MUST NOT be appended until all AIDs in the Backer Remove, `br` list have been removed.

Configuration traits field

The Configuration Traits, `c` field value is a list of strings. These are specially defined strings. Each string represents a configuration trait for the KEL. The following table defines the configuration traits. Some configuration traits MUST only appear in the Inception (delegated or not) for the KEL. Others MAY appear in either the inception event or rotation events (delegated or not). This is indicated in the third column. A Validator of an event MUST invalidate, i.e., drop any events that do not satisfy the constraints imposed by their configuration traits. If two conflicting configuration traits appear in the same list, the latter trait supersedes the earlier one.

Trait	Title	Inception Only	Description
`EO`	Establishment-Only	True	Only establishment events MUST appear in this KEL
`DND`	Do-Not-Delegate	True	This KEL MUST NOT act as a delegator of delegated AIDs
`DID`	Delegate-Is-Delegator	True	Treat a delegated AID the same as its Delegator AID
`RB`	Registrar-Backers	False	The backer list MUST provide registrar backer AIDs
`NRB`	No-Registrar-Backers	False	Registrar backers are no longer allowed

The `Establishment-Only`, `EO` config trait enables the Controller to increase its KELs security by not allowing interaction (non-establishment) events. This means all events MUST be signed by first-time, one-time pre-rotated keys. Key compromise is not possible due to repeated exposure of signing keys on Interaction events. A Validator MUST invalidate, i.e., drop any non-establishment events.

The `Do-Not-Delegate`, `DND` config trait enables the Controller to limit delegations entirely or limit the depth to which a given AID can delegate. This prevents spurious delegations. A delegation seal MAY appear in an Interaction event. Interaction events are less secure than rotation events so this configuration trait prevents delegations. In addition, a Delegatee holds its own private keys. Therefore, a given Delegatee could delegate other AIDS via interaction events that do not require the approval of its delegate. A Validator MUST invalidate, i.e., drop any delegated events whose Delegator has this configuration trait.

The `Delegate-Is-Delegator`, `DID` config trait enables the Controller to signal to validators that any Delegate (Delegatee) AIDs are to be treated as equivalent to the Delegator. This enables horizontal scaling of a Delegator's signing infrastructure.

The `Registrar-Backer`, `RB` config trait indicates that the Backer (witness) list in the establishment event in which this trait appears provides the AIDs of ledger registrar backers. The event MUST also include Registrar Backer Seal for each registrar backer in the list. This config trait enables a KEL to start with or switch to using registrar backers instead of witnesses.

The `No-Registrar-Backer`, `NRB` config trait indicates that the backer (witness) list in the establishment event in which this trait appears provides the AIDs of witnesses, not registrar backers. This config trait enables a KEL to switch back from using a registrar backer to using witnesses. When a KEL is not currently using registrar backers then the `NRB` config trait has no effect. The combination of the `RB` and `NRB` config traits enable a KEL to switch between using witnesses and registrar backers.

In the event that an establishment event includes both `RB` and `NRB` configuration traits in its configuration trait list, then the one that appears last in the configuration trait list is enforced, i.e., activated, and any earlier appearances are ignored.

Seal list field

The Seal, `a` (anchor) field value is a list of field maps representing Seals. These are defined in detail in the Seals Section below.

Seals

The dictionary definition of the seal is "evidence of authenticity". Seals make a verifiable, nonrepudiable commitment to an external serialized data item without disclosing the item and also enable that commitment to the external data to be bound to the key state of a

KEL at the location of the seal. This provides evidence of authenticity while maintaining confidentiality. This also enables the validity of the commitment to persist in spite of later changes to the key state. This is an essential feature for unbounded term but verifiable issuances. This also enables an endorsed issuance using one key state with later revocation of that issuance using a different key state. The order of appearance of seals in a KEL provides a verifiable ordering of the associated endorsements of that data, which can be used as a foundation for ordered verifiable transactions. Seals enable authenticatable transactions that happen externally to the KEL.

The collision resistance of a cryptographic strength digest makes it computationally infeasible for any other serialized data to have the same digest. Thus, a nonrepudiable signature on a digest of serialized data is equivalent to such a signature on the serialized data itself. Because all key events in a KEL are signed by the controller of that KEL, the inclusion of a seal in a key event is equivalent to signing the external data but without revealing that data. When given the external data, a Validator can verify that the seal is a digest of that data and hence verify the equivalent nonrepudiable commitment. A seal, at a minimum, includes a cryptographic digest of the serialized external data, usually its SAID. The external data MAY itself be composed of digests of other data.

Seals MAY also be used as attachments to events to provide a reference for looking up the key state to be used for signatures on that event. For seals that do not appear as attachments, the semantics of these seals are modified by the context in which the seal appears, such as appearing in the seal list of a key event in a KEL.

When a seal appears in a message whose serialization KIND is not CESR native, i.e. is one of JSON, CBOR, or MGPK then it MUST be encoded as a field map in the message's serialization format.

When a seal appears either as an attachment to a message of any serialization kind or inside a message whose serialization KIND is CESR native, then the seal MUST be encoded using CESR with the appropriate CESR count (group) code for that type of seal. There are two count codes for each seal type, one is the small-sized code and the other the big-sized code. A given count code can contain multiple seals of the same type. The count code frames all the seals in the counted group by counting the total number of quadlets (triplets) in the counted frame. Each field in a given seal is unlabeled, but its purpose is determined by its order of appearance in the set of fields for that seal. The field values appear sequentially in a CESR serialization of that seal. The field ordering is provided in the tables below.

Seal Count Codes

Code	Name	Type	Field Labels	Description
`-Q`	DigestSealSingles	SealDigest	`[d]`	Cryptographic digest as seal
`--Q`	BigDigestSealSingles	SealDigest	`[d]`	Cryptographic digest as seal
`-R`	MerkleRootSealSingles	SealRoot	`[rd]`	Merkle tree root digest as seal
`--R`	BigMerkleRootSealSingles	SealRoot	`[rd]`	Merkle tree root digest as seal
`-S`	SealSourceCouples	SealTrans	`[s, d]`	Source event issuance/delegation/transaction as seal implied AID
`--S`	BigSealSourceCouples	SealTrans	`[s, d]`	Source event issuance/delegation/transaction as seal implied AID
`-T`	SealSourceTriples	SealEvent	`[i, s, d]`	Source key event as seal
`--T`	BigSealSourceTriples	SealEvent	`[i, s, d]`	Source key event as seal
`-U`	SealSourceLastSingles	SealLast	`[i]`	Source AID last Est Event as seal

Code	Name	Type	Field Labels	Description
`--U`	BigSealSourceLastSingles	SealLast	`[i]`	Source AID last Est Event as seal
`-V`	BackerRegistrarSealCouples	SealBack	`[bi, d]`	Backer AID metadata digest as seal
`--V`	BigBackerRegistrarSealCouples	SealBack	`[bi, d]`	Backer AID metadata digest as seal
`-W`	TypedDigestSealCouples	SealBack	`[t, d]`	Typed digest as seal
`--W`	BigTypedDigestSealCouples	SealBack	`[t, d]`	Typed digest as seal

Digest seal

The value of this seal's `d` field is an undifferentiated digest of some external data item. If the data is a SAD, then the value is its SAID.

The JSON version is shown. There is also a native CESR version.

```
{
  "d": "EAU5dUws4ffM9jZjWs0QfXTnhJ1qk2u3IUhBwFVbFnt5"
}
```

Merkle Tree root digest seal

The value of this seal's `rd` field is the root of a Merkle tree of digests of external data. This enables a compact commitment to a large number of data items. A Merkle tree is constructed so that an inclusion proof of a given digest in the tree does not require disclosure of the whole tree.

The JSON version is shown. There is also a native CESR version of the seal.

```
{
  "rd": "EBMFnZjWs0QfXTnht5AU5dU9jJ1qk2u3IUhwFVbws4ff"
```

```
}
```

Source Event seal

Source event seals bind to an event that implies what AID is associated with the seal from the context in which that seal appears. This provides an implicit approval or endorsement of that event. The associated event might be an issuance, a delegation, or a transaction event.

The `s` field value is the sequence number of the associated event being sealed. It is in lower case hexidecimal text with no leading zeros. The `d` field value is the SAID of the associated event. The fields in an Event seal MUST appear in the following order `[ s, d]`.

Source event seals MAY be used for endorsing transaction events that appear in issuance/revocation registries for ACDCs. They MAY be used as references to delegating events in the KEL of the delegator where the delegator AID is implied by the context in which the seal appears.

The JSON version is shown. There is also a CESR native version of the seal.

```
{
  "s": "e",
  "d": "EJFxtbr9WioIkzTfVX4iC6Axxyg8jjKSX0ZrJgoNHiB-"
}
```

Key Event seal

Key Event seals bind an event from some other (external) KEL or other type of event log to the event in the KEL where the seal appears. This provides an implicit approval or endorsement of that external event. The `i` field value is the AID of the external event log. The `s` field value is the sequence number of the event in the external event log. It is in lower case hexidecimal text with no leading zeros. The `d` field value is the SAID of the external event. The fields in an Event seal MUST appear in the following order `[ i, s, d]`.

Event seals are used for endorsing delegated events and for endorsing external issuances of other types of data. The JSON version is shown. There is also a CESR native version of the seal.

```
{

  "i": "'EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur'",
  "s": "1",
  "d": "ENl9GdcDY-4hlg5GtVw0g2E9X7JHw-7Dr5Zq5KNirISF"
}
```

Latest establishment event seal

The latest establishment event seal's function is similar to the key event seal above except that it does not designate a specific key event but merely designates the latest establishment event in the external KEL for the AID given as its `i` field value. This seal endorses, approves or commits to the key state of the latest establishment event of the referenced KEL. This is useful for endorsing a message.

The JSON version is shown. There is also a native CESR version of the seal.

```
{
  "i": "EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur",
}
```

Registrar Backer seal

When a ledger backer or Backers are used as a secondary root-of-trust instead of a Witness pool, then a backer seal is REQUIRED. The backer registrar is responsible for anchoring key events as transactions on the ledger. In addition to the backer seal, the establishment event that designates the backer MUST also include a configuration trait (see below) of `RB` for registrar backers. This indicates that the KEL is ledger registrar-backed instead of witness pool-backed.

The `bi` field value in the seal is the non-transferable identifier of the registrar backer (backer identifier). The first seal appearing in the seal list containing the event whose `bi` field matches that registrar backer identifier is the authoritative one for that registrar (in the event that there are multiple registrar seals for the same `bi` value).

The `d` field value in the seal MUST be the SAID of the associated metadata SAD that provides the backer registrar metadata. The SAD MAY appear as the value of the seal data, `sd` field in an associated bare, `bar` message (defined later). The nested `d` SAID of this `sd` block in the bare message MUST be the `d` field in the associated seal. This metadata could include the address used to source events onto the ledger, a service endpoint for the ledger registrar, and a corresponding ledger oracle.

To reiterate, the seal MUST appear in the same establishment event that designates the registrar backer identifier as a backer identifier in the event's backer's list along with the config trait `RB`.

The JSON version is shown. There is also a native CESR version of the seal.

```
{
  "bi": "EDeCPBTHAt75Acgi9PfEciHFnc1r2DKAno3s9_QIYrXk",
  "d": "EA8_fj-Ezin_Us_gUcg5JQJkIIBnrcZt3HEIuH-E1lpe"
}
```

Typed seal

Typed seals bind some data via a digest of that data to whatever context in which the seal appears. This provides an implicit approval or endorsement of that data. The type field provides a seal type and version so that various types of digests with different semantics and derivations may be used. This allows each type to also be versioned. This provides a generic facility for seals.

The `t` field value is the versioned type of the seal. The `d` field value is a cryptographic digest of cryptographic strength. While the CESR encoding of the `d` field value provides the cryptographic algorithm used to compute the digest, the `t` field provides other information useful to its derivation. For example, there are various types of Merkle trees. In every case, the root digest is the digest of its connected nodes; the construction of the tree and proofs of inclusion and exclusion, however, may differ based on the type of Merkle tree, such as a Sparse Merkle tree versus a non-sparse tree.

The value of the type `t` field is a CESR encoded Primitive in the Text domain (qb64). It has 4 characters for the seal digest type, and 3 Base64 characters for version. The first version character is the major version, and the last two characters are the minor version. This provides for a total of 64 major versions and 4096 minor versions for each major version. For example, `CAB` represents a major version of `2` and a minor version of `01`. In dotted decimal notation, this would be `2.1`. The 4 character encoded seal type plus the 3 character encoded seal version consume 7 Base64 characters. The CESR primitive code for such a 7-character primitive is `Y`. An example seal type/version field value for seal type `CSMT` version 2.0 is as follows:

`YCSMTCAA`

The fields in a typed MUST appear in the following order `[ t, d]`.

The JSON version is shown. There is also a CESR native version of the seal.

> NOTE
>
> Examples in this section are not cryptographically verifiable

```json
{
  "t": "YCSMTCAA",
  "d": "EAU5dUws4ffM9jZjWs0QfXTnhJ1qk2u3IUhBwFVbFnt5"
}
```

Key event messages

A Key event message types MUST be one of the following `[icp, rot, ixn, dip, drt]`.

The convention for field ordering within a message is to put the fields that are common to all Message types first followed by fields that are not common. The common fields are `v`, `t`, and `d` in that order. A Validator MAY drop any provided key event message body that does not have at least one attached signature from the current controlling key state of the AID of the associated KEL.

In the following examples, for each of the Key event message types, the serialization kind used is JSON. This is JSON without whitespace, which in Python can be generated as follows:

```
import json

raw = json.dumps(sad, separators=(",", ":"), ensure_ascii=False).
encode()
```

where `sad` is a Python `dict` (field map) that includes a SAID field and therefore is self-addressed data or a self-addressed dict, i.e., a SAD. The serialization kind directly influences the value generated for the SAID (digest) of a given message.

The Annex titled “Working Examples Setup” provides more detail on how to replicate the working examples.

Inception Event Message Body

The top-level fields of an Inception, `icp`, event message body MUST appear in the following order: `[ v, t, d, i, s, kt, k, nt, n, bt, b, c, a]`. All are REQUIRED. No other top-level fields are allowed (MUST NOT appear). Signatures and other information, when attached, MUST be attached to the Message body using CESR attachment codes.

Inception Event Example

The message body is provided as a Python dict. This dict is then serialized using the SAID protocol to generate the SAIDive values in the `d` and `i` fields. The serialization kind is JSON.

```
{
    "v": "KERICAACAAJSONAAKp.",
    "t": "icp",
    "d": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "s": "0",
    "kt": "2",
    "k":
    [
        "DBFiIgoCOpJ_zW_OO0GdffhHfEvJWb1HxpDx95bFvufu",
        "DG-YwInLUxzVDD5z8SqZmS2FppXSB-ZX_f2bJC_ZnsM5",
        "DGIAk2jkC3xuLIe-DI9rcA0naevtZiKuU9wz91L_qBAV"
```

```
    ],
    "nt": "2",
    "n":
    [
        "ELeFYMmuJb0hevKjhv97joA5bTfuA8E697cMzi8eoaZB",
        "ENY9GYShOjeh7qZUpIipKRHgrWcoR2WkJ7Wgj4wZx1YT",
        "EGyJ7y3TlewCW97dgBN-4pckhCqsni-zHNZ_G8zVerPG"
    ],
    "bt": "3",
    "b":
    [
        "BGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3B",
        "BJfueFAYc7N_V-zmDEn2SPCoVFx3H20alWsNZKgsS1vt",
        "BAPv2MnoiCsgOnklmFyfU07QDK_93NeH9iKfOy8V22aH",
        "BA4PSatfQMw1lYhQoZkSSvOCrE0Sdw1hmmniDL-yDtrB"
    ],
    "c": ["DID"],
    "a": []
}
```

The raw JSON serialization of the message body is shown as a compact (no whitespace) Python byte string as follows:

```
(b'{"v":"KERICAACAAJSONAAKp.","t":"icp","d":"EPR7FWsN3tOM8PqfMap2
FRfF4MFQ4v3ZXj'
b'BUcMVtvhmB","i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhm
B","s":"0","kt":'
b'"2","k":["DBFiIgoCOpJ_zW_OO0GdffhHfEvJWb1HxpDx95bFvufu","DG-YwI
nLUxzVDD5z8Sq'
b'ZmS2FppXSB-ZX_f2bJC_ZnsM5","DGIAk2jkC3xuLIe-DI9rcA0naevtZiKuU9w
z91L_qBAV"],"'
b'nt":"2","n":["ELeFYMmuJb0hevKjhv97joA5bTfuA8E697cMzi8eoaZB","EN
Y9GYShOjeh7qZ'
b'UpIipKRHgrWcoR2WkJ7Wgj4wZx1YT","EGyJ7y3TlewCW97dgBN-4pckhCqsni-
zHNZ_G8zVerPG'
b'"],"bt":"3","b":["BGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3
B","BJfueFAYc7N'
b'_V-zmDEn2SPCoVFx3H20alWsNZKgsS1vt","BAPv2MnoiCsgOnklmFyfU07QDK_
93NeH9iKfOy8V'
b'22aH","BA4PSatfQMw1lYhQoZkSSvOCrE0Sdw1hmmniDL-yDtrB"],"c":["DI
D"],"a":[]}')
```

The next key digests in the message body are derived from the following set of next keys:

```
[
    "DLv9BlDvjcZWkfPfWcYhNK-xQxz89h82_wA184Vxk8dj",
```

```
    "DCx3WypeBym3fCkVizTg18qEThSrVnB63dFq2oX5c3mz",
    "DO0PG_ww4PbF2jUIxQnlb4DluJu5ndNehp0BTGWXErXf"
]
```

The AID created by this inception event is the value of the `i` field, that is, `EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB`. For the purposes of the examples, let this be given the user-friendly alias `ean` as in Ean's AID. Notice that the config trait list for Ean has the config trait `DID` for `Delegate-Is-Delegator` which means that Validators may treat Delegates (Delegatees) of Ean as if they were Ean.

Delegated Inception Event Message Body

The top-level fields of a Delegated Inception, `dip` event message body MUST appear in the following order: `[ v, t, d, i, s, kt, k, nt, n, bt, b, c, a, di]`. All are REQUIRED. No other top-level fields are allowed (MUST NOT appear). Signatures and other information, when attached, MUST be attached to the Message body using CESR attachment codes.

Interaction Event Message Body

The top-level fields of an Interaction, `ixn` event message body MUST appear in the following order: `[ v, t, d, i, s, p, a]`. All are REQUIRED. No other top-level fields are allowed (MUST NOT appear). Signatures and other information, when attached, MUST be attached to the Message body using CESR attachment codes.

Interaction Event Example

The message body is provided as a Python dict. This dict is then serialized using the SAID protocol to generate the SAIDive value in the `d` field. The serialization kind is JSON.

```
{
    "v": "KERICAACAAJSONAAE8.",
    "t": "ixn",
    "d": "EDeCPBTHAt75Acgi9PfEciHFnc1r2DKAno3s9_QIYrXk",
    "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "s": "1",
    "p": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "a":
    [
        {
            "i": "EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur",
            "s": "0",
            "d": "EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur"
        }
    ]
}
```

The raw JSON serialization of the message body is shown as a compact (no whitespace) Python byte string as follows:


```
(b'{"v":"KERICAACAAJSONAAE8.","t":"ixn","d":"EDeCPBTHAt75Acgi9PfE
ciHFnc1r2DKAno'
b'3s9_QIYrXk","i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhm
B","s":"1","p":"'
b'EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB","a":[{"i":"EHqSsH
1Imc2MEcgzEor'
b'dBUFqJKWTcRyTz2GRc2SG3aur","s":"0","d":"EHqSsH1Imc2MEcgzEordBUF
qJKWTcRyTz2GR'
b'c2SG3aur"}]}')
```


Notice that in this example the Issuer is Ean and uses Ean's AID for the `i` field value. This Interaction event is sealing the delegation of Fay's AID by virtue of a SealEvent dict in the data attribute field list. In the event seal, the `i` field value is Fay's AID. The `s` field value is the hex encoded sequence number of Fay's delegated inception event, and the `d` field value is the SAID of Fay's delegated inception event. The combination of the appearance of this seal in Ean's KEL for an establishment event in the KEL of a delegated AID that designates Ean's AID as the Delegator in that KEL's Inception event provides a cryptographically verifiable two-way binding (sometimes called a two-way peg) between the delegating and delegated events.

Rotation Event Message Body

The top-level fields of a Rotation, `rot` event message body MUST appear in the following order: `[ v, t, d, i, s, p, kt, k, nt, n, bt, br, ba, c, a]`. All are REQUIRED. No other top-level fields are allowed (MUST NOT appear). Signatures and other information, when attached, MUST be attached to the Message body using CESR attachment codes.

Rotation Event Example

The message body is provided as a Python dict. This dict is then serialized using the SAID protocol to generate the SAIDive value in the `d` field. The serialization kind is JSON.

```
{
    "v": "KERICAACAAJSONAAMf.",
    "t": "rot",
    "d": "EJOnAKXGaSyJ_43kit0V806NNeGWS07lfjybB1UcfWsv",
    "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "s": "2",
    "p": "EDeCPBTHAt75Acgi9PfEciHFnc1r2DKAno3s9_QIYrXk",
    "kt": "2",
    "k":
    [
```

```
        "DLv9BlDvjcZWkfPfWcYhNK-xQxz89h82_wA184Vxk8dj",
        "DCx3WypeBym3fCkVizTg18qEThSrVnB63dFq2oX5c3mz",
        "DO0PG_ww4PbF2jUIxQnlb4DluJu5ndNehp0BTGWXErXf"
    ],
    "nt": "2",
    "n":
    [
        "EA8_fj-Ezin_Us_gUcg5JQJkIIBnrcZt3HEIuH-E1lpe",
        "EERS8udHp2FW89nmaHweQWnZz7I8v9FTQdA-LZ_amqGh",
        "EAEzmrPusrj4CDKnSFQvhCEW6T95C7hBeFtZtRD7rOTg"
    ],
    "bt": "4",
    "br":
    [
        "BA4PSatfQMw1lYhQoZkSSvOCrE0Sdw1hmmniDL-yDtrB"
    ],
    "ba":
    [
        "BO3cCAfQiqndZBBxwNk6RGkyA-OA1XbZhBj3s4-VIsCo",
        "BPowpltoeF14nMbU1ng89JSoYf3AmWhZ50KaCaVO6SIW"
    ],
    "c": [],
    "a":
    [
        {
            "i": "EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur",
            "s": "1",
            "d": "ENl9GdcDY-4hlg5GtVwOg2E9X7JHw-7Dr5Zq5KNirISF"
        }
    ]
}
```


The raw JSON serialization of the message body is shown as a compact (no whitespace) Python byte string as follows:


```
(b'{"v":"KERICAACAAJSONAAMf.","t":"rot","d":"EJOnAKXGaSyJ_43kit0V
806NNeGWS07lfj'
b'ybB1UcfWsv","i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhm
B","s":"2","p":"'
b'EDeCPBTHAt75Acgi9PfEciHFnc1r2DKAno3s9_QIYrXk","kt":"2","k":["DL
v9BlDvjcZWkfP'
b'fWcYhNK-xQxz89h82_wA184Vxk8dj","DCx3WypeBym3fCkVizTg18qEThSrVnB
63dFq2oX5c3mz'
b'","DO0PG_ww4PbF2jUIxQnlb4DluJu5ndNehp0BTGWXErXf"],"nt":"2","n":
["EA8_fj-Ezin'
b'_Us_gUcg5JQJkIIBnrcZt3HEIuH-E1lpe","EERS8udHp2FW89nmaHweQWnZz7I
8v9FTQdA-LZ_a'
b'mqGh","EAEzmrPusrj4CDKnSFQvhCEW6T95C7hBeFtZtRD7rOTg"],"b
```

```
t":"4","br":["BA4PSa'
b'tfQMw1lYhQoZkSSvOCrE0Sdw1hmmniDL-yDtrB"],"ba":["BO3cCAfQiqndZBB
xwNk6RGkyA-0A'
b'1XbZhBj3s4-VIsCo","BPowpltoeF14nMbU1ng89JSoYf3AmWhZ50KaCaVO6SI
W"],"c":[],"a"'
b':[{"i":"EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3au
r","s":"1","d":"ENl9GdcDY'
b'-4hlg5GtVw0g2E9X7JHw-7Dr5Zq5KNirISF"}]}')
```

The next key digests in the message body are derived from the following set of next keys:

```
[
    "DHODGNuxeW2JTKn3S7keooAjVw582puHoK_zDflPflZg",
    "DImP4vghHKJIgzBxt1HrTLrNLOMy07_gFV0_IekdzAQh",
    "DNlPrQ9T7G71BDgRSpB0coMFANpw_QPVEUosPep1JC79"
]
```

Notice that in this example, the Issuer is Ean and uses Ean's AID for the `i` field value. This Rotation event is sealing the delegation of Fay's Rotation event by virtue of a SealEvent dict in the data attribute field list. In the event seal, the `i` field value is Fay's AID. The `s` field value is the hex encoded sequence number of Fay's delegated rotation event at hex number `1`, and the `d` field value is the SAID of Fay's delegated rotation event. The combination of the appearance of this seal in Ean's KEL for an establishment event in the KEL of a delegated AID that designates Ean's AID as the Delegator in that KEL's Inception event provides a cryptographically verifiable two-way binding (sometimes called a two-way peg) between the delegating and delegated events.

Delegated Inception Event Message Body

The top-level fields of a Delegated Inception, `dip` event message body MUST appear in the following order: `[ v, t, d, i, s, kt, k, nt, n, bt, b, c, a, di]`. All are REQUIRED. No other top-level fields are allowed (MUST NOT appear). Signatures and other information, when attached, MUST be attached to the Message body using CESR attachment codes.

Delegated Inception Event Example

The message body is provided as a Python dict. This dict is then serialized using the SAID protocol to generate the SAIDive values in the `d` and `i` fields. The serialization kind is JSON.

```
{
    "v": "KERICAACAAJSONAAL4.",
    "t": "dip",
```

```
    "d": "EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur",
    "i": "EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur",
    "s": "0",
    "kt": ["1/2", "1/2", "1/2"],
    'k':
    [
        'DEE-HCMSwqMDkEBzlmUNmVBAGIinGu7wZ5_hfY6bSMz3',
        'DHyJFyFzuD5vvUWv5jy6nwWI3wZmSnoePu29tBR-jXkv',
        'DN3JXVEvIjTbisPC4maYQWy6eQIRNdJsxqGFXYUm_ygr'
    ],
    "nt": ["1/2", "1/2", "1/2"],
    "n":
    [
        "EFzr1nnfHpT-nkSfd6vQvbPC-Kq6zy8vbVvUmwxcM1e-",
        "EIXFsLk9kmESy0ZsoHMUaDyK_g3DVRiJQYiAlyeCeYJM",
        "EGVvq4Njkki3EZv838rJrYShBtwXY9o8RUrG2w3nbujn"
    ],
    "bt": "3",
    "b":
    [
        "BFATArhqG_ktVCRLWt2Knbc7JDpaPAFJ4npNEmIW_gPX",
        "BOtF-I9geAUjX9NW1kLIq5qDRNgEXCuwpE4mKHkYuWsF",
        "BEzZUvashpXh_nfPoR6aiqvag0a8E_tbhpeJIgHhOXzl",
        "BCE6biH4a-Zg8LI3cMSx7JRoOvb8rRD62xbyl9N4M2g6"
    ],
    "c": [],
    "a": [],
    "di": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB"
}
```


The raw JSON serialization of the message body is shown as a compact (no whitespace) Python byte string as follows:


```
(b'{"v":"KERICAACAAJSONAAL4.","t":"dip","d":"EHqSsH1Imc2MEcgzEord
BUFqJKWTcRyTz2'
b'GRc2SG3aur","i":"EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3au
r","s":"0","kt":'
b'["1/2","1/2","1/2"],"k":["DEE-HCMSwqMDkEBzlmUNmVBAGIinGu7wZ5_hf
Y6bSMz3","DHy'
b'JFyFzuD5vvUWv5jy6nwWI3wZmSnoePu29tBR-jXkv","DN3JXVEvIjTbisPC4ma
YQWy6eQIRNdJs'
b'xqGFXYUm_ygr"],"nt":["1/2","1/2","1/2"],"n":["EFzr1nnfHpT-nkSfd
6vQvbPC-Kq6zy'
b'8vbVvUmwxcM1e-","EIXFsLk9kmESy0ZsoHMUaDyK_g3DVRiJQYiAlyeCeYJ
M","EGVvq4Njkki3'
b'EZv838rJrYShBtwXY9o8RUrG2w3nbujn"],"bt":"3","b":["BFATArhqG_ktV
CRLWt2Knbc7JD'
b'paPAFJ4npNEmIW_gPX","BOtF-I9geAUjX9NW1kLIq5qDRNgEXCuwpE4mKHkYuW
```

```
sF","BEzZUvas'
b'hpXh_nfPoR6aiqvag0a8E_tbhpeJIgHhOXzl","BCE6biH4a-Zg8LI3cMSx7JRo
Ovb8rRD62xbyl'
b'9N4M2g6"],"c":[],"a":[],"di":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZX
jBUcMVtvhmB"}')
```

The next key digests in the message body are derived from the following set of next keys:

```
[
    "DB1S8zOh4_qdFhxVHn7BDZb1ErWbBFvcVJX1suKSBctR",
    "DDCDFlbG4dCAX6oIbNffB1mkZqLAS_eHnYUUIPH7BeXB",
    "DP3GAMcSx7eCApzk1N7DceV42o1dZemAe0s3r_-Z0zs1"
]
```

The AID created by this inception event is the value of the `i` field, that is, `EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur`. For the purposes of the examples, let this be given the user-friendly alias `fay` as in Fay's AID. Notice that the Delegator AID given by the `di` field value is Ean's AID. To clarify, Ean is the Delegator of Fay the Delegatee. Also, notice that the thresholds for the signing keys and next rotation keys use the syntax of a fractionally weighted threshold instead of a simple numeric threshold.

Delegated Rotation Event Message Body

The top-level fields of a Delegated Rotation, `drt` event message body MUST appear in the following order: `[ v, t, d, i, s, p, kt, k, nt, n, bt, br, ba, c, a]`. All are REQUIRED. No other top-level fields are allowed (MUST NOT appear). Signatures and other information, when attached, MUST be attached to the Message body using CESR attachment codes. Notice that the Delegated Rotation event does not have a Delgator AID, `di` field. It uses the Delegator AID provided by the associated Delegated Inception event's Delegator AID, `di` field. Even though the field set is the same for both Rotation and Delegated Rotation events, the message body type of `drt` signals to Validators that the event must be validated using the rules for delegated events.

Delegated Rotation Event Example

The message body is provided as a Python dict. This dict is then serialized using the SAID protocol to generate the SAIDive value in the `d` field. The serialization kind is JSON.

```
{
    "v": "KERICAACAAJSONAAKh.",
    "t": "drt",
    "d": "ENl9GdcDY-4hlg5GtVw0g2E9X7JHw-7Dr5Zq5KNirISF",
    "i": "EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur",
    "s": "1",
```

```
    "p": "EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur",
    "kt": ["1/2", "1/2", "1/2"],
    "k":
    [
        "DB1S8zOh4_qdFhxVHn7BDZb1ErWbBFvcVJX1suKSBctR",
        "DDCDFlbG4dCAX6oIbNffB1mkZqLAS_eHnYUUIPH7BeXB",
        "DP3GAMcSx7eCApzk1N7DceV42o1dZemAe0s3r_-Z0zs1"
    ],
    "nt": ["1/2", "1/2", "1/2"],
    "n":
    [
        "EKUlc5Ml4HLSvdk39k_vh0m6rc061mfM1a4qoEuiBwXW",
        "EJdqHiijmjII-ZtlhFAM5D7myuNeESQkzHoqeWJMMHzW",
        "EDyk8pj0YPHjGNfrG2qZI866WwevwlHEbWYMsKGTGqj2"
    ],
    "bt": "3",
    "br": ["BOtF-I9geAUjX9NW1kLIq5qDRNgEXCuwpE4mKHkYuWsF"],
    "ba": ["BOMrYd5izsqbqaq1WZYa3nbEeTYLPwccfqfhirybKKqx"],
    "c": [],
    "a": []
}
```

The raw JSON serialization of the message body is shown as a compact (no whitespace) Python byte string as follows:


```
(b'{"v":"KERICAACAAJSONAAKh.","t":"drt","d":"ENl9GdcDY-4hlg5GtVwO
g2E9X7JHw-7Dr5'
b'Zq5KNirISF","i":"EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3au
r","s":"1","p":"'
b'EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur","kt":["1/2","1/
2","1/2"],"k":['
b'"DB1S8zOh4_qdFhxVHn7BDZb1ErWbBFvcVJX1suKSBctR","DDCDFlbG4dCAX6o
IbNffB1mkZqLA'
b'S_eHnYUUIPH7BeXB","DP3GAMcSx7eCApzk1N7DceV42o1dZemAe0s3r_-Z0zs
1"],"nt":["1/2'
b'","1/2","1/2"],"n":["EKUlc5Ml4HLSvdk39k_vh0m6rc061mfM1a4qoEuiBw
XW","EJdqHiij'
b'mjII-ZtlhFAM5D7myuNeESQkzHoqeWJMMHzW","EDyk8pj0YPHjGNfrG2qZI866
WwevwlHEbWYMs'
b'KGTGqj2"],"bt":"3","br":["BOtF-I9geAUjX9NW1kLIq5qDRNgEXCuwpE4mK
HkYuWsF"],"ba'
b'":["BOMrYd5izsqbqaq1WZYa3nbEeTYLPwccfqfhirybKKqx"],"c":[],"a":
[]}')
```


The next key digests in the message body are derived from the following set of next keys:

```
[
    "DCcN7BGPo6c47EWOTvcIUCpzvetDN5E-7EPMprN6tqVI",
    "DAaAPS7IpPe9nPrgF6eGkA9hIphUIeZE0zLkGHCS1BBD",
    "DONoZ4RumKezgod8xoAtRQvmhPRe4LZm8QP-BVEN-MW_"
]
```

Notice that in addition to rotating the signing keys, this event also rotates out one of the witnesses and replaces it with a new witness.

Receipt Messages

Receipt Message Body

The Receipt Message types MUST be as follows `[rct]`.

The top-level fields of a Receipt, `rct` message body MUST appear in the following order: `[ v, t, d, i, s]`. All are REQUIRED. No other top-level fields are allowed (MUST NOT appear). Signatures and Seals MUST be attached to the Message body using CESR attachment codes. The Signatures or Seals are on the Key event indicated by the top-level fields of the Receipt, not the Receipt message body itself.

The SAID, `d` field value is the SAID of a key event from a KEL, i.e., the key event being receipted, not the receipt message itself.

The Identifier AID, `i` field value is the Controller AID of the KEL for the key event being receipted.

The Sequence Number, `s` field value is the Sequence Number (hex-encoded) of the key event being receipted.

Receipt example:

The message body is provided as a Python dict. This dict is then serialized. The serialization kind is JSON. Note that unlike the other message bodies a reciept does not have a field that is the SAID of its serialization. A Receipt message body merely holds a reference to the SAID of some Key event message body.

```
{
    "v": "KERICAACAAJSONAACT.",
    "t": "rct",
    "d": "EJOnAKXGaSyJ_43kit0V806NNeGWS07lfjybB1UcfWsv",
    "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "s": "2"
}
```

The raw JSON serialization of the message body is shown as a compact (no whitespace) Python byte string as follows:

```
(b'{"v":"KERICAACAAJSONAACT.","t":"rct","d":"EJOnAKXGaSyJ_43kit0V
806NNeGWS07lfj'
b'ybB1UcfWsv","i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhm
B","s":"2"}')
```

Notice that this is a receipt for Ean's Rotation Event message above.

Routed Messages

The Routed Messages MUST include a route, `r` field, and MAY include a return route, `rr` field. The value of the route and return route fields are hierarchical. The Routed Message types MUST be as follows `[qry, rpy, pro, bar, xip, exn]`.

Routed Services

Routed messages enable a backend to employ routed services in support of KELs, KERLs, service endpoints, and supporting data for KERI. Using hierarchical routes to manage services is a powerful paradigm for both externally and internally facing APIs. By abstracting the route concept so that it is not tied to the narrow confines of ReST URL-based APIs and combining that abstraction with OOBIs that map transport schemes to AIDs, a KERI implementation can use routing across its distributed infrastructure as a unifying architectural property.

For example, once a message has been received at a transport-specific port and the appropriate authentication (secure attribution) policy has been applied, it can be forwarded to a service router that distributes the message to the process that handles it. One way to effect that distribution is to prefix the external route provided in the message with an internal route that redirects the message appropriately. Thus, routing can affiliate the external-facing API with any internal-facing API. A return route enables the response to be returned despite asynchronous internal processing of the request. With this approach, no artificial synchronous state needs to be maintained to match outgoing and incoming messages. The internal routes can reflect different types of routing, such as intra-process, inter-process, inter-host, inter-protocol, and inter-database.

A given implementation could have multiple types of routers, each with different properties, including security properties.

Routing Security

Suppose that some information needs to be protected as sealed-confidential where sealed means the information is bound to the KEL via a Seal and confidential means that the information is sensitive and needs to be protected. A KEL conveys two types of information:

- information that is public to the KEL, namely key state. In general, key state includes not just the current signing keys but all the associated information including thresholds for both signing keys, next pre-rotated key digests, witness pool identifiers and threshold and configuration data. Any viewer of a KEL can view this key state. Thus, the publicity of the KEL itself determines the publicity of its key state data. Other public information can be sealed to a KEL. The seal itself is a cryptographic digest that does not disclose the data. Still, if the associated data is provided elsewhere in a public manner, then the seal provides no confidentiality protection but merely a verifiable binding. An example of this type of data is a transaction event log used for a revocation registry for an ACDC.
- information that is hidden but sealed to the key state in the KEL. A seal includes a cryptographic digest of information. The presence of the seal in an event in the KEL binds that data to the key state at that event but without disclosing the information. Thus the binding is public but the information is not. When the information includes sufficient cryptographic entropy, such as through the inclusion of a salty-nonce (UUID) then an attacker can not discover that data even with a rainbow table attack. The digest effectively hides or blinds the data. This enables the data to be protected or treated as sensitive or confidential. Access to the KEL does not disclose the data. Some other exchange process is required to disclose or un-blind the data. This type is appropriate for sealed confidential information.

One security vulnerability of routed architectures is attacks on the routers themselves (especially router configuration, both static and dynamic). This vulnerability is most acute when a single router needs to handle information with different security properties. One solution to this problem is to use a pre-router that can redirect messages to different post-routers with different security properties. For example, a pre-router would route sensitive data to a sensitive data post-router and non-sensitive data to a non-sensitive data post-router. This ensures that sensitive and non-sensitive data are never mixed. This enables tighter, more secure configuration control over data flows within an infrastructure. The best pre-routers act early in the routing process.

In KERI, the earliest possible place for a pre-router is at the stream parser. The stream parser does not look at routes but does look at message types. Therefore, a stream parser as a pre-router needs the sensitive data to be segregated by message type. As a result, the KERI protocol supports two classes of routed messages distinguished by message types. The first class is denoted by query-reply-exchange messages, and the second by prod-bare messages (see the appropriate sections below for a description of the message types). The first class, query-reply-exchange, can be used for the first type of information above, namely, information public to a KEL. The second class, prod-bare, can be used for the second type of information, namely, hidden but sealed to a KEL (sealed confidential). When a given implementation chooses to use one router for both classes of information, it needs to take appropriate measures to protect the router.

Notably, the exchange message types are only associated with the first class of data. This is because exchange messages are signed by the participating peers but not sealed. Once an exchange transaction is completed successfully, the set of messages in that transaction can be aggregated and then sealed to the participating peer's KELs. The transaction set can then be treated as sealed-confidential information, and its subsequent disclosure is managed with prod-bare messages. An exchange message can reference a data item that is sealed but the disclosure of that seal can happen with a bare, `bar` message. Often, the point of an exchange is to negotiate a chain-link confidential disclosure of information. The detailed disclosure can happen out-of-band to the exchange that negotiates the contractual commitments to that data. Those commitments use cryptographic digests that maintain confidentiality. Later disclosure of the information can be facilitated with a prod-bare pair.

Reserved field labels in routed messages

Reserved field labels in other KERI message body types:

Label	Title	Description
`v`	Version String	enables regex parsing of field map in CESR stream
`t`	Message Type	three character string
`d`	Digest SAID	fully qualified digest of block in which it appears
`u`	UUID Salty Nonce	fully qualified universally unique random salty nonce
`i`	Identifier Prefix (AID)	fully qualified primitive of the Controller (sender) AID
`ri`	Receiver Identifier Prefix (AID)	fully qualified primitive of the message Receiver (recipient) AID
`x`	Exchange SAID	fully qualified unique digest for an exchange transaction
`p`	Prior SAID	fully qualified digest, prior message SAID
`dt`	Issuer relative ISO date/time string	
`r`	Route	delimited path string for routing message
`rr`	Return Route	delimited path string for routing a response (reply or bare) message
`q`	Query Map	field map of query parameters
`a`	Attribute Map	field map of message attributes

Unless otherwise clarified below, the definitions of the `[v, t, d, i]' field values are the same as found above in the Key Event message body section.

UUID Salty Nonce field

The UUID `u` field value is a cryptographic strength salty nonce, i.e., a random number with approximately 128 bits of entropy. This ensures that it is universally unique. This universal uniqueness also ensures that the SAID of the enclosing message will also be universally unique, in spite of all other fields having the same values as some other message, but a different UUID field value. This field appears in exchange transaction inception messages to ensure that the associated transaction ID is also universally unique.

Controller AID field

The Controller AID, `i` field value is an AID that controls its associated KEL. When the Controller Identifier AID, `i` field appears at the top-level of a Routed Message, it refers to the Controller AID of the sender of that message. A Controller AID, `i` field MAY appear in other places in messages (not at the top level). In those cases, its meaning SHOULD be determined by the context of its appearance.

Receiver AID field

The Receiver AID, `ri` field value is an AID of the receiver (recipient) of an exchange message. The receiver is a controller on its associated KEL but is not the sender of the exchange message. The Reciever Identifier AID, `ri` field appears at the top-level of a Routed Exchange Message, it refers to the AID of the receiver (recipient) of that message.

Prior event SAID field

The prior, `p` field is the SAID of the prior exchange message in a transaction. When the prior `p` field appears in an exchange message, its value MUST be the SAID of the immediately preceding exchange message in that transaction. When an exchange message is not part of a transaction, then the prior `p` field value MUST be the empty string.

Exchange identifier field

The Exchange Identifier SAID, `x` field value MUST be the SAID, `d` field value of the first message in the set of exchange messages that constitute a transaction. The first message MUST be an Exchange Inception message with type `xip`. The SAID, `d` field value of the Exchange Inception message is strongly bound to the details of that message. As a cryptographic strength digest, it is a universally unique identifier. Therefore, the appearance of that value as the Exchange identifier, the `x` field in each subsequent exchange message in a transaction set, universally uniquely associates them with that set. Furthermore, the prior `p` field value in each subsequent exchange message verifiably orders the transaction set in a duplicity-evident way. When an exchange message is not part of a transaction, the Exchange Identifier, `x` field value, MUST be the empty string.

Datetime, `dt` field

The datetime, `dt` field value, if any, MUST be the ISO-8601 datetime string with microseconds and UTC offset as per IETF RFC-3339. In a given field map (block) the primary datetime will use the label, `dt`. The usage context of the message and the block where a given DateTime, `dt` field appears determines which clock (sender or Receiver) the datetime is relative to.

An example datetime string in this format is as follows:

`2020-08-22T17:50:09.988921+00:00`

Route field

The Route, `r` field value is a '/' delimited string that forms a path. This indicates the target of a given message that includes this field. This enables the message to replicate the function of the path in a ReST resource. When used in an Exchange Transaction Inception, `xip` or Exchange, `exn` message, the Route, `r` field value defines both the type of transaction and a step within that transaction. For example, suppose that the route path head value, `/ipex/` means that the transaction type is an issuance and presentation exchange transaction and the full route path value, `/ipex/offer` means that the message is the `offer` step in such a transaction.

Return Route field

The Return Route, `rr` field value is a '/' delimited string that forms a path. This allows a message to indicate how to target the associated response to the message. This enables messages on asynchronous transports to associate a given response with the message that triggered the response.

Query field

The Query, `q` field value is a field map (block). Its fields provide the equivalent of query parameters in a ReST resource.

Attribute field

The Attribute, `a` field value is a field map (block). Its fields provide the attributes conveyed by the message.

Query Message Body

The top-level fields of a Query, `qry` message body MUST appear in the following order: `[ v, t, d, dt, r, rr, q]`. All are REQUIRED. No other top-level fields are allowed (MUST NOT appear). Signatures and Seals MUST be attached to the Message body using CESR attachment codes.

Example Query Message

The message body is provided as a Python dict. This dict is then serialized using the SAID protocol to generate the SAIDive value in the `d` field. The serialization kind is JSON.

```
{
    "v": "KERICAACAAJSONAAEe.",
    "t": "qry",
    "d": "EEiUK4cVgcyA1Dk6g2jFzqc5JerkaSnJi3IosutVCyYO",
    "i": "EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur",
    "dt": "2025-08-21T17:50:00.000000+00:00",
    "r": "/oobi",
    "rr": "/oobi/process",
    "q":
    {
        "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
        "role": "witness"
    }
}
```

The raw JSON serialization of the message body is shown as a compact (no whitespace) Python byte string as follows:

```
(b'{"v":"KERICAACAAJSONAAEe.","t":"qry","d":"EEiUK4cVgcyA1Dk6g2jF
zqc5JerkaSnJi3'
b'IosutVCyYO","i":"EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3au
r","dt":"2025-08'
b'-21T17:50:00.000000+00:00","r":"/oobi","rr":"/oobi/proces
s","q":{"i":"EPR7FW'
b'sN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB","role":"witness"}}')
```

The issuer/sender of this message is Fay because Fay's AID appears in the top-level `i` field. The query could be to fetch a witness OOBI for Ean. This is determined by the route and the appearance of Ean's AID in the `i` field of the query `q` block and the role being "witness".

Reply Message Body

The top-level fields of a Reply, `rpy` message body MUST appear in the following order: `[ v, t, d, dt, r, a]`. All are REQUIRED. No other top-level fields are allowed (MUST NOT appear). Signatures and Seals MUST be attached to the Message body using CESR attachment codes.

Example Reply Message

The message body is provided as a Python dict. This dict is then serialized using the SAID protocol to generate the SAIDive value in the `d` field. The serialization kind is JSON.

```
{
    "v": "KERICAACAAJSONAAFR.",
    "t": "rpy",
    "d": "EPdgmUkvx5o_KRg3elBqj_vSZOFgWI9hCVWO-FfGZz8U",
    "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "dt": "2020-08-21T17:52:00.000000+00:00",
    "r": "/oobi/process",
    "a":
    {
        "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
        "url": "https://example.com/witness/BGKV6v93ue5L5wsgk75t6
j8TcdgABMN9x-eIyPi96J3B"
    }
}
```

The raw JSON serialization of the message body is shown as a compact (no whitespace) Python byte string as follows:

```
(b'{"v":"KERICAACAAJSONAAFR.","t":"rpy","d":"EPdgmUkvx5o_KRg3elBq
j_vSZOFgWI9hCV'
b'WO-FfGZz8U","i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhm
B","dt":"2020-08'
b'-21T17:52:00.000000+00:00","r":"/oobi/process","a":{"i":"EPR7FW
sN3tOM8PqfMap'
b'2FRfF4MFQ4v3ZXjBUcMVtvhmB","url":"https://example.com/witness/B
GKV6v93ue5L5w'
b'sgk75t6j8TcdgABMN9x-eIyPi96J3B"}}')
```

Ean is the issuer/sender of this reply whose data attribute block includes the url of a service endpoint (aka url OOBI) for one of Ean's witnesses.

Prod Message Body

The top-level fields of a Prod, `pro` message body MUST appear in the following order: `[ v, t, d, dt, r, rr, q]`. All are REQUIRED. No other top-level fields are allowed (MUST NOT appear). Signatures and Seals MUST be attached to the Message body using CESR attachment codes. The fundamental difference between the Prod messages, `pro`, and the identically structured Query messages, `qry`, is that the data targeted by Prod messages is Sealed data. By "Sealed," it is meant that the SAID of the data item, as a SAD (self-addressed data), is anchored with a seal in the KEL of the data source. Whereas the data targeted by Query, `qry` messages, is unconstrained.

Example Prod Message

The message body is provided as a Python dict. This dict is then serialized using the SAID protocol to generate the SAIDive value in the `d` field. The serialization kind is JSON.

```
{
    "v": "KERICAACAAJSONAAEp.",
    "t": "pro",
    "d": "EHNqhJXgUdYHFzNiu07Ue06QWRnOMjhTrVt_QGOfZjH_",
    "i": "EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur",
    "dt": "2025-08-21T17:50:00.000000+00:00",
    "r": "/confidential",
    "rr": "/confidential/process",
    "q":
    {
        "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
        "name": True
    }
}
```

The raw JSON serialization of the message body is shown as a compact (no whitespace) Python byte string as follows:

```
(b'{"v":"KERICAACAAJSONAAEp.","t":"pro","d":"EHNqhJXgUdYHFzNiu07U
e06QWRnOMjhTrV'
b't_QGOfZjH_","i":"EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3au
r","dt":"2025-08'
b'-21T17:50:00.000000+00:00","r":"/confidential","rr":"/confident
ial/process",'
b'"q":{"i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB","name":
true}}')
```

In this example, Fay is prodding Ean for his name.

Bare Message Body

The top-level fields of a Reply, `bar` message body MUST appear in the following order: `[ v, t, d, dt, r, a]`. All are REQUIRED. No other top-level fields are allowed (MUST NOT appear). Signatures and Seals MUST be attached to the Message body using CESR attachment codes. The fundamental difference between the Bare, `bar`, and the identically structured Reply, `rpy`, messages is that the data returned by Bare messages is Sealed data. By “Sealed,” it is meant that the SAID of the data item, as a SAD (self-addressed data), is anchored with a seal in the KEL of the data source… Whereas the data returned by Reply, `rpy` messages, is unconstrained.

Example Bare Message

The message body is provided as a Python dict. This dict is then serialized using the SAID protocol to generate the SAIDive value in the `d` field. The serialization kind is JSON.

```
{
    "v": "KERICAACAAJSONAAEV.",
    "t": "bar",
    "d": "EMSlSHIe04CuAqhz55nAnBpE_0T65Sqs2fmaPpsNIbnn",
    "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "dt": "2020-08-22T17:52:00.000000+00:00",
    "r": "/confidential/process",
    "a":
    {
        "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
        "name": "Ean"
    }
}
```

The raw JSON serialization of the message body is shown as a compact (no whitespace) Python byte string as follows:

```
(b'{"v":"KERICAACAAJSONAAEV.","t":"bar","d":"EMSlSHIe04CuAqhz55nA
nBpE_0T65Sqs2f'
b'maPpsNIbnn","i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhm
B","dt":"2020-08'
b'-22T17:52:00.000000+00:00","r":"/confidential/process","a":
{"i":"EPR7FWsN3tO'
b'M8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB","name":"Ean"}}')
```

This message has Ean exposing his name.

Exchange Transaction Inception Message Body

The top-level fields of an Exchange Transaction Incept, `xip` message body MUST appear in the following order: `[ v, t, d, u, i, ri, dt, r, q, a]`. All are REQUIRED. No other top-level fields are allowed (MUST NOT appear). Signatures and Seals MUST be attached to the Message body using CESR attachment codes.

Example Exchange Transaction Inception Message

The message body is provided as a Python dict. This dict is then serialized using the SAID protocol to generate the SAIDive value in the `d` field. The serialization kind is JSON.

```
{
    "v": "KERICAACAAJSONAAFn.",
    "t": "xip",
```

```
    "d": "EJbE2agA3239Iusld1lNvFAxRuhv1SX0mAxxUm67gWOU",
    "u": "0ABrZXJpc3BlY3dvcmtyYXcw",
    "i": "EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur",
    "ri": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "dt": "2020-08-30T13:30:10.123456+00:00",
    "r": "/offer",
    "q":
    {
        "timing": "immediate"
    },
    "a":
    {
        "action": "sell",
        "item": "Rembrant",
        "price": 300000.0
    }
}
```

The raw JSON serialization of the message body is shown as a compact (no whitespace) Python byte string as follows:

```
(b'{"v":"KERICAACAAJSONAAFn.","t":"xip","d":"EJbE2agA3239Iusld1lN
vFAxRuhv1SX0mA'
b'xxUm67gWOU","u":"0ABrZXJpc3BlY3dvcmtyYXcw","i":"EHqSsH1Imc2MEcg
zEordBUFqJKWT'
b'cRyTz2GRc2SG3aur","ri":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMV
tvhmB","dt":"'
b'2020-08-30T13:30:10.123456+00:00","r":"/offer","q":{"timing":"i
mmediate"},"a'
b'":{"action":"sell","item":"Rembrant","price":300000.0}}')
```

This message is an offer (ask) from Fay (sender) to sell a painting to Ean (receiver).

Exchange Message Body

The top-level fields of an Exchange, `exn` message body MUST appear in the following order: `[ v, t, d, i, ri, x, p, dt, r, q, a]`. All are REQUIRED. No other top-level fields are allowed (MUST NOT appear). Signatures and Seals MUST be attached to the Message body using CESR attachment codes.

Example Exchange Transaction Message

The message body is provided as a Python dict. This dict is then serialized using the SAID protocol to generate the SAIDive value in the `d` field. The serialization kind is JSON.

```
{
    "v": "KERICAACAAJSONAAGt.",
```

```
    "t": "exn",
    "d": "EEIp1e5v4L6rt7cp1nRsn4mN6bJVUDyIQEATIzxR8UnE",
    "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "ri": "EHqSsH1Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur",
    "x": "EJbE2agA3239Iusld1lNvFAxRuhv1SX0mAxxUm67gWOU",
    "p": "EJbE2agA3239Iusld1lNvFAxRuhv1SX0mAxxUm67gWOU",
    "dt": "2020-08-30T13:42:11.123456+00:00",
    "r": "/agree",
    "q":
    {
        "timing": "immediate"
    },
    "a":
    {
        "action": "buy",
        "item": "Rembrant",
        "price": 300000.0
    }
}
```

The raw JSON serialization of the message body is shown as a compact (no whitespace) Python byte string as follows:

```
(b'{"v":"KERICAACAAJSONAAGt.","t":"exn","d":"EEIp1e5v4L6rt7cp1nRs
n4mN6bJVUDyIQE'
b'ATIzxR8UnE","i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhm
B","ri":"EHqSsH1'
b'Imc2MEcgzEordBUFqJKWTcRyTz2GRc2SG3aur","x":"EJbE2agA3239Iusld1l
NvFAxRuhv1SX0'
b'mAxxUm67gWOU","p":"EJbE2agA3239Iusld1lNvFAxRuhv1SX0mAxxUm67gWO
U","dt":"2020-'
b'08-30T13:42:11.123456+00:00","r":"/agree","q":{"timing":"immedi
ate"},"a":{"a'
b'ction":"buy","item":"Rembrant","price":300000.0}}')
```

This message is an agreement (bid) from Ean (sender) to buy the painting from Fay (receiver). Notice that the `x` and `p` field values bind this message to the `xip` message from Fay.

Signing and sealing KERI data structures

Indexed Signatures

Cryptographic Signatures are computed on the serialization of a KERI data structure. The serializations use CESR. The signatures are also encoded in CESR and can be attached to the KERI data structure as part of a CESR stream. Signatures and other infor-

mation, when attached, MUST be attached to the Message body using CESR attachment codes.

CESR provides special indexed signature codes for signatures that index the signature to the public key inside a key list inside a KERI Establishment event message data structure. This way, only the indexed signature MUST be attached, not the public key needed to verify the signature. The public key is looked up from the index into the key list in the appropriate establishment event in the KEL. CESR also supports group codes that differentiate the type of indexed signatures in the group and enable pipelined extraction of the whole group for processing when attached [1]. Indexed signatures MAY be attached to both key event messages and non-key event messages. In this case, information about the associated key state for the signature MUST be attached. This is typically a reference to the AID, sequence number, and SAID (digest), of the establishment event that determines the key state. In other cases, that latest key state is assumed, and only the AID of the signer is REQUIRED. In the former case, where the signature is attached to a key event, the AID MAY be inferred.

There are two types of attached indexed signatures: controller-indexed and witnessed-indexed. Other information, such as the type of event to which the signature is attached and/or which AID the indexed signature belongs, MAY accompany the attachment.

Controller-indexed signatures index into either or both the signing key list from the latest Establishment event (inception, delegated inception, rotation, delegated rotation) and the rotating key digest list from the Establishment event immediately prior to the latest Establishment event (prior next key digest list) if any. Both of these lists are strictly ordered so that only the index is needed to determine the public key. Depending on the event and the key lists, a controller-indexed signature MAY REQUIRE one or two indices. Controller-indexed signatures attached to Interaction events and non-key-event messages need only one index into the current signing key list from the most recent prior establishment event. Controller-indexed signatures attached to Inception events (delegated or not) need only one index into the current signing key list in the Inception event. Controller-indexed signatures attached to Rotation events (delegated or not) MAY REQUIRE two indices, one into the signing key list of the rotation event itself and the other into the rotation key digest list from the immediately prior Establishment Event (Inception or Rotation), i.e., the prior next key digest list.

Witness-indexed signatures index into the effective Witness list as established by the latest Establishment event (interaction or rotation). To clarify, witness-indexed signatures attached to any type of key event (inception, rotation, interaction, delegated inception, delegated rotation) need only one index into the current list of witnesses that is in effect as of the latest establishment event, which MAY or MAY NOT be the event to which the witness signature is attached. Witnesses MUST use only nontransferable identifiers,

which include the controlling public key. Consequently, the public key needed for witness signature verification can be extracted from the witness identifier given by the effective witness list.

CESR codes for indexed signatures support up to two indices per code so that, at most, one encoded instance of an indexed signature needs to be attached. The first index in a given code provides an offset into the signing list in the message to which the indexed signature is attached. The signing list may be the signing key list when it is controller-indexed or the witness AID list when it is witness-indexed. The second index in a given code, when present, is always controller-indexed and is an offset into the prior next key digest list. The CESR group code used to attach the indexed signature MAY vary depending on the type of key event or other type of message.

Recall that a prior next key digest MUST be exposed as a public key in the succeeding rotation event signing key list when used to sign. Therefore, when the second index is present, it is used to look up the public key digest from the prior next key digest list, then the first index is used to look up the exposed public key from the signing key list, then the digest is verified against the exposed public key, and finally, the doubly indexed signature is verified using the exposed public key. Verification of the digest means digesting the exposed public key using the same digest type as the prior next key digest and comparing the digests.

A set of controller-indexed signatures on an interaction or inception event (delegated or not) MUST at least satisfy the current signing threshold in order for that event to be accepted as valid.

A set of controller-indexed signatures on a non-key event message (see below) MUST at least satisfy the signing threshold for the establishment event indicated by the event reference in the attachment group (which MAY or MAY NOT be the current signing threshold) to be accepted as valid.

A set of controller-indexed signatures on a rotation event (delegated or not) MUST at least satisfy both the current signing threshold and the prior next rotation threshold in order for that event to be accepted as valid.

A set of witness-indexed signatures on an interaction, inception, or rotation (delegated or not) for which the effective witness list is not empty MUST satisfy the current witness threshold (of accountable duplicity) for that event to be accepted as valid.

Events that have a non-empty set of attached signatures which set does not satisfy the REQUIRED thresholds SHOULD escrow the event while waiting for other signatures to arrive either as attachments to the same Version of the event or to a receipt of that event (see next section). A Validator that receives a key event or non-key-event message that does not have attached at least one verifiable Controller signature MUST drop that message (i.e., not escrow or otherwise accept it). This protects the Validator from a DDoS attack with spurious unsigned messages.

Indexed signatures minimize the space requirements for signatures. The indexed signatures codes are provided in the CESR code table for indexed signatures [1]. Given an indexed signature, a Validator looks up the associate public key from the index into the appropriate table.

Non-indexed signatures

CESR also supports codes for Signatures that are not indexed. In this case, additional information MUST be attached, such as the associated public key, in order for a validator to verify the signature. This additional information MUST be in the form of a CESR group defined by a CESR group code. [1]

Endorsements

Other entities, as identified by their AIDs, can decide to attach signatures on key events for a KEL where the signer's AID is neither the controlling AID of that KEL nor a witness AID of that event in that KEL. These non-controller, non-witness signatures can be called Endorsements. For example, a Watcher, when replaying events from a KEL its watches could choose to attach its own signature to the event in order to endorse or otherwise commit to that version of the event as the one the Watcher has seen. In this case, the attachment MUST include at least the AID of the endorser as well as any other information needed by the Validator to securely attribute the signature to its source and key state. CESR provides group codes for attaching signature Endorsements for both transferable and non-transferable AIDs with indexed and non-indexed signatures as applicable (see CESR table).

Sealing

Any serialized data can be sealed in a KEL and thereby bound to the associated key state by including the associated seal in a key event. Seals MUST include a cryptographic digest or digest proof of the serialized data. This SHOULD be the SAID of the data when that data follows the SAID protocol, i.e., is a SAD [1]. This enables later verification of the sealing when given the data. Because all events in a KEL are signed by the KEL's controller, a seal, once bound or anchored via inclusion in an event, represents an indirect signature on the sealed data. One property of cryptographic strength digests is cryptographic strength collision resistance. Such resistance makes it computationally infeasible for any two distinct (non-identical) data items to have the same digest. Therefore, a commitment via a nonrepudiable signature on a cryptographic strength digest of a data item is equivalent to a signature on the data item itself. Sealing, therefore, provides a type of indirect endorsement. The notable advantage of a seal as an indirect endorsement over a direct endorsement signature is that the seal is also bound to the key state of the endorser at the location in the KEL where the seal appears. This enables the validity of the endorsement to persist in spite of later changes to the key state. This is an essential feature for unbounded term - but verifiable - issuances. This also enables an endorsed issuance using one key state with later revocation of that issuance using a differ-

ent key state. The order of appearance of seals in a KEL provides a verifiable ordering of the associated endorsements of that data, which can be used as a foundation for ordered verifiable transactions.

One primary use case for sealing in KERI is delegated AIDs. The Delegator (AID) approves (endorses) the associated delegation of a delegated event in the Delegatee's KEL by sealing the SAID of that delegated event in the Delegator's KEL. Because the Delegator signs the sealing event, the presence of the delegated event's SAID (cryptographic digest) in the Delegator's KEL is equivalent cryptographically to a signed endorsement by the Delegator of the delegated event itself but with the added advantage that the validity of that delegation persists in spite of changes to the key state of the Delegator. A validator need only receive an attached reference to the delegating event that includes the seal in order to look up the seal and verify its presence. CESR provides codes for attached event seal references as well as codes for event seals.

Receipt Signatures

Receipt message data structures are not key events but merely reference key events (see below). A signature attached to a Receipt is not a signature on the serialized receipt data structure but is a signature on a serialization of the referenced key event. This enables the asynchronous receipt and processing of any type of signature, including controller-indexed and witness-indexed signatures. The Validator first looks up an already received copy of the referenced serialized key event and then verifies the signatures as if they had been attached to the event. Because Receipts can be more compact than the full event, they allow more efficient asynchronous distribution of signatures for events. A Validator that receives a Receipt for an event that the Validator has not yet received can escrow the Receipt and its attached signatures. This escrow, however, could be vulnerable to a DDoS attack due to spurious event references.

Receipt Seals

Similar to attached signatures, a Receipt message can convey an attached seal reference that allows a validator to associate the sealing event in the sealer's KEL with the reference to the sealed event given by the Receipt body. CESR provides codes for attached seal source references to receipts. [1]

KERI key management

KERI keypair labeling convention

In order to make key event expressions both clearer and more concise, a keypair labeling convention is used. When an AID's Key state is dynamic, i.e., the set of controlling keypairs is transferable, then the keypair labels are indexed in order to represent the successive sets of keypairs that constitute the Key state at any position in the KEL. Specifically, indexes on the labels for AIDs that are transferable to indicate which set of

keypairs is associated with the AID at any given point in its Key state or KEL. In contrast, when the Key state is static, i.e., the set of controlling keypairs is non-transferable, then no indexes are needed because the Key state never changes.

Recall that a keypair is a two tuple (public, private) of the respective public and private keys in the keypair. For a given AID, the labeling convention uses an uppercase letter label to represent that AID. When the Key state is dynamic, a superscripted index on that letter is used to indicate which keypair is used at a given Key state. Alternatively, the index MAY be omitted when the context defines which keypair and which Key state, such as, for example, the latest or current Key state. To reiterate, when the Key state is static, no index is needed.

In general, without loss of specificity, an uppercase letter label is used to represent both an AID and, when indexed, to represent its keypair or keypairs that are authoritative at a given Key state for that AID. In addition, when a keypair is expressed in tuple form `(A, a)`, the uppercase letter represents the public key, and the lowercase letter represents the private key. For example, let `A` denote the AID as a shorthand, let `A` also denote the current keypair, and finally, let the tuple `(A, a)` also denote the current keypair. Therefore, when referring to the keypair itself as a pair and not the individual members of the pair, either the uppercase label, `A`, or the tuple, `(A, a)`, MAY be used to refer to the keypair itself. When referring to the individual members of the keypair then the uppercase letter `A`, refers to the public key, and the lowercase letter `a`, refers to the private key.

The sequence of keypairs that are authoritative (i.e., establish control authority) for an AID MUST be indexed by the zero-based integer-valued, strictly increasing by one, variable 'i'. The associated indexed keypair is denoted (A^i, a^i). Furthermore, as described above, an Establishment event MAY change the Key state (usually). The sequence of Establishment events SHOULD be indexed by the zero-based integer-valued, strictly increasing by one, variable `j`. the associated key pair is denoted $(A^{i,j}, a^{i,j})$ When the set of controlling keypairs that are authoritative for a given Key state includes only one member, then `i = j` for every keypair, and only one index is needed. But when the set of keypairs used at any time for a given Key state includes more than one member, then `i != j` for every keypair, and both indices are needed.

Case in which only one index is needed

Because `i = j`, the indexed keypair for AID, A, is denoted by A^i or in tuple form by (A^i, a^i) where the keypair that is indexed uses the ith keypair from the sequence of all keypairs.

Example of the keypair sequence - one index where each keypair is represented only by its public key:

Expressed as the list, $[A^0, A^1, A^2, \ldots]$

Where: A^0 is the zero element in this sequence; (A^0, a^0) is the tuple form.

Case in which both indexes are needed

Because `i != j`, the indexed keypair for AID, `A`, is denoted by $A^{i,j}$ or in tuple form by $(A^{i,j}, a^{i,j})$ where the keypair that is indexed is authoritative or potentially authoritative as the ith keypair from the sequence of all keypairs that is authoritative in the jth Key state.

Example of the keypair sequence – two indices using three keypairs at each key state where each keypair is represented only by its public key: Expressed as the list, $[A^{0,0}, A^{1,0}, A^{2,0}, A^{3,1}, A^{4,1}, A^{5,1}]$.

Where: the first two Key states will consume the first six keypairs of the list.

Labelling the digest of the public key

With pre-rotation, each public key from the set of pre-rotated keypairs MUST be hidden as a qualified cryptographic digest of that public key. The digest of the public key labeled `A` is represented using the functional notation `H(A)` for hash (digest).

Example of a singly indexed digest - A^i is denoted by $H(A^i)$

Example of a doubly indexed digest - $A^{i,j}$ is denoted by $H(A^{i,j})$

Where:

The digest of the public key labeled `A` is represented using the functional notation `H(A)` for hash (digest).

A pre-rotated keypair is potentially authoritative for the next or subsequent Establishment event after the Establishment event when the digest of the pre-rotated keypair first appears. Therefore, its jth index value is one greater than the jth index value of the Establishment event in which its digest first appears. Let `j` represent the index of the jth Establishment event, then the pre-rotated digests appearing in that Establishment event have index `j+1`. As explained in more detail for partial Rotation of a pre-rotated set, a pre-rotated keypair from a set of two or more pre-rotated keypairs is only potentially authoritative so that its actual authoritative j^{th} index MAY change when it is actually rotated in, if ever.

Example of public key and the pre-rotated digest of the next public key - $[A^{0,0}], [H(A^{1,1})]$

Labelling key events in a KEL

Finally, each Key event in a KEL MUST have a zero-based integer-valued, strictly increasing by one, sequence number represented by the variable `k`. Abstractly, the variable `k` can be used as an index on any keypair label to denote the sequence number of an event for which that keypair is authoritative. Usually, this appears as a subscript. Thus any given keypair label could have three indices, namely, `i,j,k`. A public key would appear as $A^{i,j}_k$.

Example of labeling key events in a KEL - $A^{i,j}_{k}$

Where:

`i` denotes the ith keypair from the sequence of all keypairs; `j` denotes the jth Establishment event in which the keypair is authoritative; and `k` denotes the sequence number of kth Key event in which the keypair is authoritative.

Expressed as a sequence of lists of two public keys per event:

$[A^{0,0}_{0}, A^{1,0}_{0}], [A^{0,0}_{1}, A^{1,0}_{1}], [A^{0,0}_{2}, A^{1,0}_{2}], [A^{2,1}_{3}, A^{3,1}_{3}], [A^{2,1}_{4}, A^{3,1}_{4}]$

When a KEL has only Establishment events, then `j = k`. In that case, either `j` or `k` is redundant.

Example of public keys from KEL with only establishment events - Expressed as a sequence of lists of two public keys per event where `j` is omitted because `j = k`:

$[A^{0}_{0}, A^{1}_{0}], [A^{2}_{1}, A^{3}_{1}]$

Pre-rotation

Each Establishment event involves two sets of keys that each play a role that together establish complete control authority over the AID associated with the location of that event in the KEL. To clarify, control authority is split between keypairs that hold signing authority and keypairs that hold rotation authority. A Rotation revokes and replaces the keypairs that hold signing authority, as well as replacing the keypairs that hold rotation authority. The two sets of keys are labeled current and next. Each Establishment event designates both sets of keypairs. The first (current) set consists of the authoritative signing keypairs bound to the AID at the location in the KEL where the Establishment event occurs. The second (next) set consists of the pre-rotated authoritative rotation keypairs that will be actualized in the next (ensuing) Establishment event. Each public key in the set of next (ensuing) pre-rotated public keys is hidden in or blinded by a digest of that key. When the Establishment event is the Inception event, then the current set is the initial set. The pre-rotated next set of Rotation keypairs are one-time use only rotation keypairs, but MAY be repurposed as signing keypairs after their one-time use to rotate.

In addition, each Establishment event designates two threshold expressions, one for each set of keypairs (current and next). The Current threshold determines the needed satisficing subset of signatures from the associated current set of keypairs for signing authority to be considered valid. The Next threshold determines the needed satisficing subset of signatures from the associated next set of hidden keypairs for rotation authority to be considered valid. The simplest type of threshold expression for either threshold is an integer that is no greater than nor no less than the number of members in the set. An integer threshold acts as an `M of N` threshold where `M` is the threshold and `N` is the total number of keypairs represented by the public keys in the key list. If any set of `M` of

the `N` private keys belonging to the public keys in the key list verifiably signs the event, then the threshold is satisfied by the Controller exercising its control authority role (signing or rotating) associated with the given key list and threshold.

To clarify, each Establishment event MUST include a list (ordered) of the qualified public keys from each of the current (initial) set of keypairs, a threshold for the current set, a list (ordered) of the qualified cryptographic digests of the qualified public keys from the next set of keypairs, and a threshold for the next set. Each event MUST also include the AID itself as either a qualified public key or a qualified digest of the Inception event.

Each Non-establishment event MUST be signed by a threshold-satisficing subset of private keys from the current set of keypairs from the most recent Establishment event. The following sections detail the requirements for a valid set of signatures for each type of Establishment event.

Inception event pre-rotation

The creator of the Inception event MUST create two sets of keypairs, the current (initial) set and the next set. The private keys from the current set are kept as secrets. The public keys from the current set are exposed via inclusion in the Inception event. Both the public and private keys from the next set are kept as secrets, and only the cryptographic digests of the public keys from the next set are exposed via inclusion in the event. The public keys from the next set are only exposed in a subsequent Establishment event, if any. Both thresholds are exposed via inclusion in the event.

Upon emittance of the Inception event, the current (initial) set of keypairs becomes the current set of Verifiable authoritative signing keypairs for the AID. Emittance of the Inception event also issues the identifier. Moreover, to be verifiably authoritative, the Inception event MUST be signed by a threshold-satisficing subset of the current (initial) set of private keys. The Inception event MUST be verified against the attached signatures using the included current (initial) list of public keys. When self-addressing, a digest of the serialization of the Inception event provides the AID itself as derived by the SAID protocol SAID.

There MUST be only one Establishment event that is an Inception event. All subsequent Establishment events MUST be Rotation events.

When the AID in the `i` field is a SAID, the new Inception event has two qualified digest fields. In this case, both the `d` and `i` fields MUST have the same value. This means the digest suite's derivation code, used for the `i` field MUST be the same for the `d` field. The derivation of the `d` and `i` fields is special. Both the `d` and `i` fields are replaced with dummy `#` characters of the length of the digest to be used. The digest of the Inception event is then computed and both the `d` and `i` fields are replaced with the

qualified digest value. Validation of an Inception event requires examining the `i` field's derivation code and if it is a digest-type then the `d` field MUST be identical otherwise the Inception event is invalid.

When the AID is not self-addressing, i.e., the `i` field derivation code is not a digest, then the `i` is given its value and the `d` field is replaced with dummy characters `#` of the correct length and then the digest is computed, which is the standard SAID algorithm.

Rotation using pre-rotation

Unlike Inception, the creator of a Rotation event MUST create only one set of keypairs, the newly next set. Both the public and private keys from the newly created next set are kept as secrets and only the cryptographic digests of the public keys from the newly next set are exposed via inclusion in the event. The list of newly current public keys MUST include the old next threshold-satisficing subset of old next public keys from the most recent prior Establishment event. For short, the next threshold from the most recent prior Establishment event is denoted as the prior next threshold, and the list of unblinded public keys taken from the blinded key digest list from the most recent prior Establishment event as the prior next key list. The subset of old prior next keys that are included in the newly current set of public keys MUST be unhidden or unblinded because they appear as the public keys themselves and no longer appear as digests of the public keys. Both thresholds are exposed via inclusion in the event.

Upon emittance of the Rotation event, the new current keypairs become the current set of Verifiable authoritative signing keypairs for the identifier. The old current set of keypairs from the previous Establishment event has been revoked and replaced by the newly current set. Moreover, to be verifiably authoritative, the rotation event MUST be signed by a dual threshold-satisficing subset of the newly current set of private keys., meaning that the set of signatures on a Rotation event MUST satisfy two thresholds. These are the newly current threshold and the old prior next threshold from the most recent prior Establishment event. Therefore, the newly current set of public keys MUST include a satisfiable subset with respect to the old prior next threshold of public keys from the old prior next key list. The included newly current list of public keys enables verification of the Rotation event against the attached signatures.

The act of including the digests of the new next set of public keys in each Establishment event essentially performs a pre-rotation operation on that set by making a Verifiable forward-blinded commitment to that set. Consequently, no other key set can be used to satisfy the threshold for the next Rotation operation. Because the next set of pre-rotated keys is blinded (i.e., has not been published or otherwise exposed, i.e., used to sign), an attacker cannot forge and sign a Verifiable Rotation operation without first unblinding the pre-rotated keys. Therefore, given sufficient cryptographic strength of the digests, the only attack surface available to the adversary is a side-channel attack on the private key store itself and not on signing infrastructure. But the Controller, as the creator of the

pre-rotated private keys is free to make that key store as arbitrarily secure as needed because the pre-rotated keys are not used for signing until the next Rotation. In other words, as long as the creator keeps secret the pre-rotated public keys themselves, an attacker must attack the key storage infrastructure because side-channel attacks on signing infrastructure are obviated.

As explained in the First seen policy section, for a Validator, the first seen rule applies, that is, the first seen Version of an event is the authoritative one for that Validator. The first seen wins. In other words, the first published Version becomes the first seen. Upon Rotation, the old prior next keys are exposed but only after a new next set has been created and stored. Thus, the creator is always able to stay one step ahead of an attacker. By the time a new Rotation event is published, it is too late for an attacker to create a Verifiable Rotation event to supplant it because the original Version has already been published and could be first seen by the Validator. The window for an attacker is the network latency for the first published event to be first seen by the network of Validators. Any later, key compromise is too late.

In essence, each key set follows a Rotation lifecycle where it changes its role with each Rotation event. A pre-rotated keypair set starts as the member of the next key set holding one-time rotation control authority. On the ensuing Rotation, that keypair becomes part of the current key set holding signing control authority. Finally on the following Rotation, that keypair is discarded. The lifecycle for the initial key set in an Inception event is slightly different. The initial key set starts as the current set holding signing authority and is discarded on the ensuing Rotation event, if any.

Pre-Rotation example:

Recall that the keypairs for a given AID can be represented by the indexed letter label such as $A^{i,j}_{k}$ where ‘i’ denotes the ith keypair from the sequence of all keypairs, 'j’ denotes the jth Establishment event in which the keypair is authoritative, and 'k’ represents the kth Key event in which the keypair is authoritative. When a KEL has only Establishment events, then j = k. When only one keypair is authoritative at any given Key state then i = j.

Also, recall that a pre-rotated keypair is designated by the digest of its public key appearing in an Establishment event. The digest is denoted as H(A) or $H(A^{i,j}_{k})$ in indexed form. The appearance of the digest makes a forward Verifiable cryptographic commitment that can be realized in the future when and if that public key is exposed and listed as a current authoritative signing key in a subsequent Establishment event.

The following example illustrates the lifecycle roles of the key sets drawn from a sequence of keys used for three Establishment events; one Inception followed by two Rotations. The initial number of authoritative keypairs is three and then changes to two and then changes back to three.

Event	Current Keypairs	CT	Next Keypairs	NT
0	[$A^{0,0}$, $A^{1,0}$, $A^{2,0}$]	2	[H($A^{3,1}$), H($A^{4,1}$)]	1
1	[$A^{3,1}$, $A^{4,1}$]	1	[H($A^{5,2}$), H($A^{6,2}$), H($A^{7,2}$)]	2
2	[$A^{5,2}$, $A^{6,2}$, $A^{7,2}$]	2	[H($A^{8,3}$), H($A^{9,3}$), H($A^{10,3}$]	2

Where:

CT means Current threshold. NT means Next threshold.

Fractionally weighted threshold

A simple fractionally weighted threshold consists of a list of one or more clauses where each clause is a list of rational fractions (i.e., ratios of non-negative integers expressed as fractions, where zero is not allowed in the denominator). Each entry in each clause in the fractional weight list corresponds one-to-one to a public key appearing in a key list in an Establishment event. Key lists order a key set. A list of clauses acts to order the set of rational fraction weights that appear in the list of clauses. Satisfaction of a fractionally weighted threshold requires satisfaction of every clause in the list. In other words, the clauses are logically ANDed together. Satisfaction of any clause requires that the sum of the weights in that clause that corresponds to verified signatures on that event MUST sum to at least a weight of one. Using rational fractions and rational fraction summation avoids the problem of floating-point rounding errors and ensures the exactness and universality of threshold satisfaction computations. In a complex fractionally weighted threshold, each weight in a clause MAY be either a simple weight or a weighted list of weights. This provides an additional layer of nesting of weights.

For example, consider the following simple single clause fractionally weighted threshold, [1/2, 1/2, 1/2]. Three weights mean there MUST be exactly three corresponding key pairs. Let the three keypairs in one-to-one order be denoted by the list of indexed public keys, [A^0, A^1, A^2]. The threshold is satisfied if any two of the public keys sign because 1/2 + 1/2 = 1. This is exactly equivalent to an integer-valued ‘2 of 3’ threshold.

The public key’s appearance order in a given key list and its associated threshold weight list MUST be the same.

Fractionally weighted thresholds become more interesting when the weights are not all equal or include multiple clauses. Consider the following five-element single clause fractionally weighted threshold list, [1/2, 1/2, 1/2, 1/4, 1/4] and its corresponding public key list, [A^0, A^1, A^2, A^3, A^4]. Satisfaction would be met given signatures from any two or more of A^0, A^1, or A^2 because each of these keys has a weight of 1/2 and the combination of any two or more sums to 1 or more. Alternatively, satisfaction would be met with signatures from any one or more of A^0, A^1, or A^2 and both of A^3, and A^4 because any of those combinations would sum to 1 or more. Because participation of A^3 and A^4 is not required as long as at least two of A^0, A^1, and A^2 are available then A^3 and A^4 MAY be treated as reserve members of the controlling set of keys. These reserve members only need to participate if only one of the other three is available. The flexibility of a fractionally weighted threshold enables redundancy in the combinations of keys needed to satisfy both day-to-day and reserve contingency use cases.

As a complex example, consider the following threshold with some weights as nested weighted lists of weights.

```
[[{"1/2": ["1/2", "1/2", "1/2"]}, "1/2", {"1/2": ["1", "1"]}],
["1/2", {"1/2": ["1", "1"]}]]
```

The corresponding public key list has 9 entries, as follows:

[A^0, A^1, A^2, A^3, A^4, A^5, A^6, A^7, A^0].

There are two clauses in the threshold.

The first clause is as follows:

```
[{"1/2": ["1/2", "1/2", "1/2"]}, "1/2", {"1/2": ["1", "1"]}]
```

The first element is a nested weighted list of three weights expressed as a field map or JSON object block with only one field. The key or label for this field is the weight on the list of weights, and the value of this field is the list of three weights as a nested fractionally weighted threshold. Therefore, this first element corresponds to the three entries A^0, A^1, and A^2 in the key list. Because each entry in this list has the same weight, 1/2, valid signatures for any two of A^0, A^1, and A^2 will satisfy the nested threshold. Satisfaction of the nested threshold contributes the weight, 1/2 that the field label provides to the satisfaction of the enclosing clause.

The second element is a simple weight of 1/2, corresponding to A^3 in the key list.

The third element is a nested weighted list of two weights that correspond to A^4 and A^5 in the key list. Because each of the nested weights is 1, signatures from only one of A^4 and A^5 need to be valid to satisfy the nested threshold. Satisfaction contributes the field key weight of 1/2 to the clause.

Suppose that for the first clause, valid signatures from the following keys are provided: A^0, A^3, and A^5. Then, the following weights are contributed for the clause: 1/2 and 1/2. These sum to 1, so the clause threshold is satisfied. Note that because only one of the three weights for the first element was satisfied, the first element did not contribute any weight to the clause satisfaction.

The second clause is as follows:

```
["1/2", {"1/2": ["1", "1"]}]
```

The first element is a simple weight of 1/2, which corresponds to A^6 in the key list.

The second element is a nested weighted list of two weights that correspond to A^7 and A^8 in the key list. Because each of the nested weights is 1, signatures from only one of A^7 and A^8 need to be valid to satisfy the nested threshold. Such satisfaction contributes the field key weight of 1/2 to the clause.

Suppose that for the first clause, valid signatures from the following keys are provided: A^6, and A^8. Then, the following weights are contributed for the clause: 1/2 and 1/2. These sum to 1, so the clause threshold is satisfied.

Given that signature from A^0, A^3, A^5, A^6, and A^8 are valid then both clauses will be satisfied so that the whole threshold is satisfied.

A use case for complex nested weighted lists of weights is when a given contributor to a fractionally weighted threshold manages its keys across multiple devices such that each device contributes a key. The total weight from the given contributor needs to be normalized to a single weight, but the satisfaction of its contribution is itself a fractionally weighted threshold across that contributor's devices.

General Pre-rotation

The KERI protocol includes support for something called Partial rotation, where not all pre-rotated keys become signing keys. A Partial rotation operation on a set of pre-rotated keys MAY hold back some as unexposed while exposing others as needed. The KERI protocol also supports Augmented rotation, where new signing keys are added that were not pre-rotated keys.

A given Rotation event may use a combination of Partial and Augmented Rotation to enable support for important use cases like reserve rotation, custodial rotation, and surprise quantum attack recovery.

As described above, a valid Rotation operation requires the satisfaction of two different thresholds:

1. The Next threshold from the given Rotation event's most recent prior Establishment event concerning its associated blinded next key digest list. For short, the next threshold from the most recent prior Establishment event is denoted as the *prior-next* threshold. The list of unblinded public keys taken from the blinded key digest list in the most recent prior Establishment event is the *prior-next* key list. Explicating the elements of the *prior-next* key list requires exposing or unblinding the underlying public keys committed to by their corresponding digests appearing in the next key digest list of the most recent prior Establishment event. For short, this is called the *prior-next* key digest list. Importantly, the unexposed (blinded) public keys from the *prior-next* key list MAY be held in reserve to be used in some later Rotation event.
2. The current threshold of the given Rotation event concerning its associated current public key list.

More precisely, any Rotation event's current public key list MUST include both a satisfiable subset of the *prior-next* key list concerning the prior next threshold and a satisfiable set of public keys concerning its current threshold.

In other words, the current public key list MUST be satisfiable concerning both the *prior-next* and current thresholds.

The *prior-next* threshold and *prior-next* key list are not to be confused with the next threshold and the next key list. To clarify, the *prior-next* threshold is provided by the most recent prior establishment event (Inception or Rotation). The next threshold, on the other hand, is provided by the current Rotation event. The next threshold is not used to verify the signatures on the current Rotation event but will be used on the next (subsequent) Rotation event.

A rotation event estabishes two control authorities: verifiable rotation control authority and verifiable signing control authority.

To establish verifiable rotation control authority, the current key list MUST include a satisfiable subset of exposed (unblinded) pre-rotated next keys from the most recent prior Establishment event. Where satisfiable concerns the prior next threshold. The exposed pre-rotated keys must be verified against their pre-committed digests from the prior-next establishment event.

To establish a Verifiable signing control authority, the current key list MUST also include a satisfiable subset of public keys. Where satisfiable concerns the current threshold.

These two conditions are satisfied trivially whenever the prior-next and current key lists and thresholds are equivalent. When both the current and the prior-next key lists and thresholds are identical, the validation can be simplified by comparing the two lists and thresholds to confirm that they are identical and then confirming that the signatures satisfy the threshold for the key list. When not identical, the Validator MUST perform the appropriate mathematical set operations to confirm compliance.

Recall that the public key's appearance order in a given key list and its associated threshold weight list MUST be the same. The order of appearance, however, of any public keys that appear in both the current and prior next key lists may be different between the two key lists and, hence, the two associated threshold weight lists. A Validator, therefore, MUST confirm that the set of keys in the current key list truly includes a satisfiable subset of the prior next key list and that the current key list is satisfiable with respect to both the current and prior next thresholds. Actual satisfaction means that the set of attached signatures MUST satisfy both prior-next and current thresholds as applied to their respective key lists.

Suppose the current public key list does not include a proper subset of the prior-next key list. This means that no keys were held in reserve. This also means that the current key list is either identical to the prior-next key list or is a superset of the prior-next key list. Nonetheless, such a Rotation may change the current key list and or threshold with respect to the prior-next key list and/or threshold as long as it meets the satisfiability constraints defined above.

If the current key list includes the full set of keys from the prior-next key list, then a full Rotation has occurred, not a Partial Rotation because no keys were held in reserve or omitted. A full Rotation may also be an Augmented rotation when it adds new keys to the current key list and/or changes the current threshold with respect to the prior next key list and threshold.

Reserve Rotation

As described above, the pre-rotation mechanism supports partial pre-rotation or, more exactly, partial Rotation of pre-rotated keypairs. One important use case for partial Rotation is to enable pre-rotated keypairs designated in one Establishment event to be held in reserve and not exposed at the next (immediately subsequent) Establishment event. This reserve feature enables keypairs held by Controllers as members of a set of pre-rotated keypairs to be used for fault tolerance in the case of non-availability by other Controllers while at the same time minimizing the burden of participation by the reserve members. In other words, a reserved pre-rotated keypair contributes to the potential availability and fault tolerance of control authority over the AID without necessarily requiring the participation of the reserve key-pair in a Rotation until and unless it is needed to provide continuity of control authority in the event of a fault (non-availability of a non-reserved member). This reserve feature enables different classes of key Controllers to contribute to the control authority over an AID. This enables provisional key control authority. For example, a key custodial service or key escrow service could hold a keypair in reserve to be used only upon satisfaction of the terms of the escrow agreement. This could be used to provide continuity of service in the case of some failure event. Provisional control authority may be used to prevent types of common-mode failures without burdening the provisional participants in the normal non-failure use cases.

Reserve rotation example:

Provided here is an illustrative example to help clarify the pre-rotation protocol, especially with regard to threshold satisfaction for Reserve rotation.

SN	Role	Keys	Threshold
0	Crnt	$[A^0, A^1, A^2, A^3, A^4]$	[1/2, 1/2, 1/2, 1/4, 1/4]
0	Next	$[H(A^5), H(A^6), H(A^7), H(A^8), H(A^9)]$	[1/2, 1/2, 1/2, 1/4, 1/4]
1	Crnt	$[A^5, A^6, A^7]$	[1/2, 1/2, 1/2]
1	Next	$[H(A^{10}), H(A^{11}), H(A^{12}), H(A^8), H(A^9)]$	[1/2, 1/2, 1/2, 1/4, 1/4]
2	Crnt	$[A^{10}, A^8, A^9]$	[1/2, 1/2, 1/2]
2	Next	$[H(A^{13}), H(A^{14}), H(A^{15}), H(A^{16}), H(A^{17})]$	[1/2, 1/2, 1/2, 1/4, 1/4]
3	Crnt	$[A^{13}, A^{14}, A^{15}]$	[1/2, 1/2, 1/2]
3	Next	$[H(A^{18}), H(A^{19}), H(A^{20}), H(A^{16}), H(A^{17})]$	[1/2, 1/2, 1/2, 1/4, 1/4]
4	Crnt	$[A^{18}, A^{20}, A^{21}]$	[1/2, 1/2, 1/2]
4	Next	$[H(A^{22}), H(A^{23}), H(A^{24}), H(A^{16}), H(A^{17})]$	[1/2, 1/2, 1/2, 1/4, 1/4]
5	Crnt	$[A^{22}, A^{25}, A^{26}, A^{16}, A^{17}]$	[1/2, 1/2, 1/2, 0, 0]
5	Next	$[H(A^{27}), H(A^{28}), H(A^{29}), H(A^{30}), H(A^{31})]$	[1/2, 1/2, 1/2, 1/4, 1/4]

Where, in the column labels:

SN is the sequence number of the event. Each event uses two rows in the table. Role is either Current (Crnt) or Next and indicates the role of the key list and threshold on that row. Keys is the list of public keys denoted with the indexed label of the keypair sequence. Threshold is the threshold of signatures that MUST be satisfied for validity.

Commentary on each event:

(0) Inception: Five keypairs have signing authority, and five other keypairs have rotation authority. Any two of the first three or any one of the first three, and both the last two are sufficient. This anticipates holding the last two in reserve.

(1) Rotation: The first three keypairs from the prior next, A^5, A^6, and A^7, are rotated at the new current signing keypairs. This exposes the keypairs. The last two from the prior next, A^8 and A^9, are held in reserve. They have not been exposed and are included in the next key list.

(2) Rotation: The prior next keypairs, A^{11} and A^{12}, are unavailable to sign the Rotation and participate as part of the newly current signing keys. Therefore, A^8 and A^9 MUST be activated (pulled out of reserve) and included and exposed as both one-time rotation keys and newly current signing keys. The signing authority (weight) of each of A^8 and A^9 has been increased to 1/2 from 1/4. This means that any two of the three of A^{10}, A^8, and A^9 may satisfy the signing threshold. Nonetheless, the Rotation event #2 MUST be signed by all three of A^{10}, A^8, and A^9 in order to satisfy the prior next threshold because in that threshold A^8, and A^9 only have a weight of 1/4.

(3) Rotation: The keypairs $H(A^{16})$, $H(A^{17})$ have been held in reserve from event #2

(4) Rotation: The keypairs $H(A^{16})$, $H(A^{17})$ continue to be held in reserve.

(5) Rotation: The keypairs A^{16}, and A^{17} are pulled out of reserve and exposed in order to perform the Rotation because A^{23}, and A^{24} are unavailable. Two new keypairs, A^{25}, A^{26}, are added to the current signing key list. The current signing authority of A^{16}, and A^{17} is none because they are assigned a weight of 0 in the new current signing threshold. For the Rotation event to be valid, it MUST be signed by A^{22}, A^{16}, and A^{17} in order to satisfy the prior next threshold for rotation authority and also MUST be signed by any two of A^{22}, A^{25}, and A^{26} in order to satisfy the new current signing authority for the event itself. This illustrates how reserved keypairs may be used exclusively for rotation authority and not for signing authority.

Custodial Rotation

Partial rotation combined with Augmented rotation supports another important use case, that of Custodial key rotation. Because control authority is split between two key sets, the first for signing authority and the second (pre-rotated) for rotation authority, the associated thresholds and key list can be structured in such a way that a designated custodial agent can hold signing authority while the original Controller can hold exclusive rotation

authority. The holder of the rotation authority can then, at any time, without the cooperation of the custodial agent, if needed, revoke the agent's signing authority and assign it to some other agent or return that authority to itself.

Custodial rotation example:

Provided here is an illustrative example to help to clarify the pre-rotation protocol, especially about threshold satisfaction for Custodial rotation.

SN	Role	Keys	Threshold
0	Crnt	$[A^0, A^1, A^2]$	[1/2, 1/2, 1/2]
0	Next	$[H(A^3), H(A^4), H(A^5)]$	[1/2, 1/2, 1/2]
1	Crnt	$[A^3, A^4, A^5, A^6, A^7, A^8]$	[0, 0, 0, 1/2, 1/2, 1/2]
1	Next	$[H(A^9), H(A^{10}), H(A^{11})]$	[1/2, 1/2, 1/2]
2	Crnt	$[A^9, A^{10}, A^{11}, A^{12}, A^{13}, A^{14}]$	[0, 0, 0, 1/2, 1/2, 1/2]
2	Next	$[H(A^{15}), H(A^{16}), H(A^{17})]$	[1/2, 1/2, 1/2]

Where for the column labels:

SN is the sequence number of the event. Each event uses two rows in the table. Role is either Current (Crnt) or Next and indicates the role of the key list and threshold on that row. Keys is the list of public keys denoted with indexed label of the keypair sequence. Threshold is the threshold of signatures that MUST be satisfied for validity.

Commentary on each event:

(0) Inception: The private keys from current signing keypairs A^0, A^1, and A^2 are held by the custodian of the identifier. The owner of the identifier provides the digests of the next rotation keypairs, $H(A^3)$, $H(A^4)$, and $H(A^5)$ to the custodian in order that the custodian may include them in the event and then sign the event. The owner holds the private keys from the next rotation keypairs A^3, A^4, and A^5. A self-addressing AID would then be created by the formulation of the Inception event. Once formed, the custodian controls the signing authority over the identifier by virtue of holding the associated private keys for the current key list. But the Controller exercises the rotation authority by virtue of holding the associated private keys for the next key list. Because the Controller of the rotation au-

thority may at their sole discretion revoke and replace the keys that hold signing authority, the owner, holder of the next private keys, is ultimately in control of the identifier so constituted by this Inception event.

(1) Rotation: The owner changes custodians with this event. The new custodian creates new current signing keypairs, A^6, A^7, and A^8 and holds the associated private keys. The new custodian provides the public keys A^6, A^7, and A^8 to the owner so that the owner can formulate and sign the Rotation event that transfers signing authority to the new custodian. The owner exposes its rotation public keys, A^3, A^4, and A^5 by including them in the new current key list. But the weights of those rotation keys in the new current signing threshold are all 0 so they have no signing authority. The owner creates a new set of next keypairs and includes their public key digests, $H(A^9)$, $H(A^{10})$, $H(A^{11})$ in the new next key list. The owner holds the associated private keys and thereby retains rotation authority. This event MUST be signed by any two of A^3, A^4, and A^5 in order to satisfy the prior next threshold and also MUST be signed by any two A^6, A^7, and A^8 in order to satisfy the new current signing threshold. The new current threshold and new next threshold clearly delineate that the new custodian now holds exclusive signing authority and owner continues to retain exclusive rotation authority.

(2) Rotation: Change to yet another custodian following the same pattern as event #1.

Surprise Quantum Attack Recovery (SQAR)

Partial rotation, together with Augmented rotation, supports another important use case, that of Surprise Quantum Attack Recovery (SQAR). In an SQAR rotation, the resultant signing authority MUST use post-quantum safe key pairs. Because it's a surprise quantum attack, it is assumed that both the current signing keys (before the SQAR rotation) and the prior-next key pairs are not post-quantum safe. This means that a surprise quantum attacker could invert the current signing public keys to discover the associated private keys and thereby forge verifiable Interaction events. Because the prior-next key digests are post-quantum safe in that a quantum computer is not advantaged over a conventional computer in inverting cryptographic strength digests. The surprise quantum attacker is not able to forge a verifiable Rotation event. It is also assumed that the first exposure of the prior-next keys occurs after the publication of the SQAR rotation to the witnesses. At which point it is now too late for the quantum attacker. Coincident with the SQAR rotation, the signing control authority is post-quantum safe. This recovers signing control authority from the compromised signing keys by rotation to post-quantum safe signing keys.

SQAR rotation example:

Provided here is an illustrative example to help clarify the pre-rotation protocol, especially regarding threshold satisfaction for the SQAR rotation.

SN	Role	Keys	Threshold
0	Crnt	[A^0, A^1, A^2]	[1/2, 1/2, 1/2]
0	Next	[H(A^3), H(A^4), H(A^5)]	[1/2, 1/2, 1/2]
1	Crnt	[A^3, A^4, A^5, A^6, A^7, A^8]	[0, 0, 0, 1/2, 1/2, 1/2]
1	Next	[H(A^9), H(A^{10}), H(A^{11})]	[1/2, 1/2, 1/2]

Where for the column labels:

SN is the sequence number of the event. Each event uses two rows in the table. Role is either Current (Crnt) or Next and indicates the role of the key list and threshold on that row. Keys is the list of public keys denoted with the indexed label of the keypair sequence. Threshold is the threshold of signatures that MUST be satisfied for validity.

Commentary on each event:

(0) Inception: The private keys from current signing keypairs A^0, A^1, and A^2 are non-post-quantum safe. The key pairs are represented by the next digests H(A^3), H(A^4), and H(A^5) are also non-post-quantum safe.

(1) SQAR Rotation: The controller recovers from a surprise quantum attack with this rotation. The new current signing keypairs, A^6, A^7, and A^8 use post-quantum-safe signing key pair generation algorithms. The controller exposes its non-quantum-safe rotation public keys, A^3, A^4, and A^5 by including them in the new current key list. But the weights of those rotation keys in the new current signing threshold are all 0, so they have no signing authority, only rotation authority. Therefore, the new signing authority is not subject to a post-quantum attack. The controller creates a new set of next keypairs that are post-quantum-safe and includes their public key digests, H(A^9), H(A^{10}), H(A^{11}) in the new next key list. Now all future Interaction events are protected with post-quantum-safe signing keys, and the next Rotation is already using post-quantum safe rotation keys.

Technically, the next rotation key digests could be for non-post-quantum safe keys as long as the next rotation was also an SQAR rotation. This is because the next key digests are post-quantum safe commitments to the next rotation keys. In an SQAR rotation, none of the rotation keys is given any current signing authority, all weights are zero.

Cooperative Delegation

A delegation or identifier delegation operation is provided by a pair of events. One event is the delegating event in the KEL of the Delegator and the other event is the delegated event in the KEL of the Delegatee. This pairing of events is a somewhat novel approach to delegation in that the resultant delegation requires cooperation between the Delegator and Delegatee. This is called cooperative delegation. In a cooperative delegation, a delegating identifier approves the establishment operation (inception or rotation) of a delegated identifier. A delegating event is a type of event that includes in its data payload an event seal of the delegated event that is the target the delegation operation. This delegated event seal includes a digest of the delegated event. This verifiably seals or anchors or binds the delegated event to the KEL of the Delegator.

Likewise, the inception event of the Delegatee's KEL includes the delegator's AID. This binds the inception and any later establishment events in the Delegatee's KEL to a unique delegator. A Validator MUST be given or find the delegating seal in the delegator's KEL before the event may be accepted as valid. The pair of bindings (delegation seal in delegator's KEL and delegator's AID in Delegatee's inception event) makes the delegation cooperative. Both MUST participate. As will be seen later, this cooperation adds an additional layer of security to the Delegatee's KEL and provides a way to recover from a pre-rotated key compromise.

Because the delegating event payload is a list, a single delegating event may perform multiple delegation operations, one for each delegation seal.

A delegation operation directly seals an establishment event for a delegated AID. Either an inception or rotation. Thus, a delegation operation either delegates an inception or a rotation that respectively, either creates or rotates the authoritative keys for delegated AID. The AID for a Delegatee AID (delegated identifier prefix) MUST be a fully qualified digest of its inception event, which includes a reference to the Delegator's AID. This cryptographically binds the Delegatee's AID to the Delegator's AID.

The Delegator (controller) retains establishment control authority over the delegated identifier in that the new delegated identifier may only authorize non-establishment events with respect to itself. Delegation, therefore, authorizes revokable signing authority to some other AID. The delegated identifier has a delegated key event sequence where the inception event is a delegated inception, and any rotation events are delegated rotation events. Control authority for the delegated identifier, therefore, requires verification of a given delegated establishment event, which in turn requires verification of the delegating identifier's establishment event.

To reiterate, because the delegating event seal includes a digest of the full delegated event, it thereby provides a cryptographic commitment to the delegated event and all of its configuration data.

A common use case of delegation would be to delegate signing authority to a new identifier prefix. The signing authority may be exercised by a sequence of revokable signing keys distinct from the keys used for the delegating identifier. This enables horizontal scalability of signing operations. The other major benefit of a cooperative delegation is that any exploiter that merely compromises only the delegate's authoritative keys may not capture the control authority of the delegate. A successful exploiter must also compromise the delegator's authoritative keys. Any exploit of the Delegatee is recoverable by the delegator. Conversely, merely compromising the delegator's signing keys may not enable a delegated rotation without also compromising the Delegatee's pre-rotated keys. Both sets of keys must be compromised simultaneously. This joint compromise requirement is a distinctive security feature of cooperative delegation. Likewise, as explained later, this cooperative feature also enables recovery of a joint compromise of a delegation at any set of delegation levels by a recovery at the next higher delegation level.

Security Properties of Pre-rotation

For many exploits, the likelihood of success is a function of exposure to continued monitoring or probing. Narrowly restricting the exposure opportunities for exploitation in terms of time, place, and method, especially if the time and place happen only once, makes exploitation extremely difficult. The exploiter has to either predict the one-time and place of that exposure or has to have continuous universal monitoring of all exposures. By declaring the very first pre-rotation in the inception event, the window for its exploit is as narrow as possible. Likewise, each subsequent rotation event is a one-time and place signing exposure of the former next (pre-rotated) rotation key.

Because each pre-rotation makes a cryptographic future commitment to a set of one-time first-time rotation keys, later exploit of the current authoritative signing key(s) may not capture key rotation authority as it has already been transferred via the pre-commitment to a new unexposed set of keys. To elaborate, the next (ensuing) pre-rotated keypairs in an inception event serve as first-time, one-time, and only-time rotation keys in the next rotation operation. Thereafter, those keypairs may be activated as the new current (root) authoritative signing key(s) but no longer have rotation authority. Likewise, the next (ensuing) pre-rotated keypairs in each rotation event serve as first-time, one-time, and only-time rotation keys in the next rotation operation. Thereafter, those keypairs may be activated as the new current (root) authoritative signing key(s) but likewise no longer have rotation authority.

In administrative identity (identifier) systems, the binding between keys, controller, and identifier may be established by administrative fiat. As a result, administrative fiat may be used as a recovery mechanism for compromised administrative keys. This may make those administrative keys relatively more exposed through multiple use of each key. In contrast, when the binding between keys, controller, and identifier is purely cryptographic (decentralized), such as is the case with this (KERI) protocol, there is no recovery mechanism once the keys for the root control authority have been fully captured. Therefore,

security over those keys is more critical. As a result, in this protocol, administrative (establishment operation) keys are first-time, one-time, and only-time use as administrative keys.

Dead-Attacks

By definition, a Dead-attack on a given establishment event occurs after the Key state for that event has become stale because a later establishment event has rotated the sets of signing and pre-rotated keys to new sets. There are two types of Dead-attacks. The first is a compromise of the stale signing keys from a stale establishment event needed to sign non-establishment events, such as an interaction event. This is denoted as a non-establishment Dead-Attack. The second is a compromise of the stale pre-rotated keys from a stale establishment event needed to sign a subsequent establishment event, such as a rotation event. This is denoted as an establishment Dead-attack.

Non-establishment Dead-attack

A successful non-establishment Dead-Attack first must compromise the set of signing keys for some past but stale interaction (non-establishment) event; second, create an alternate verifiable version of that stale interaction event; and third, propagate this alternate event to a given validator before the original event has had time to propagate to that validator or any other component the validator may access as First-seen. This looks like what is commonly known as an [24] on a validator. To protect against such an attack, a controller MUST propagate the event sufficiently widely enough that the attacker cannot eclipse all components, such as Watchers, that the validator may consult. The more distant the stale event is in the past the more difficult it becomes to mount a successful eclipse attack because the event would have more time to be universally propagated to the full network of watchers. Otherwise, the Validator would have already First-seen the original event and the compromised event would be dropped i.e., cannot be accepted as First-seen by the Validator. Network propagation times are, at most, seconds and may be as little as milliseconds, which only opens a very short time window of how stale a stale event may be before it is sufficiently protected from any such eclipse attack. Should the event also be protected with a witness pool, then the attacker must compromise not only the stale signing keys but also a threshold-satisficing number of witnesses protecting that event. This could make a non-establishment attack practically infeasible.

The one exception would be the case where the event's key state has only a single signing key and a single prior pre-rotated key that has been repurposed as the single signing key, which the signing key has been compromised. In this case, the attacker could then attempt an establishment Dead-attack by creating a compromised state rotation event using the stale compromised signing key as a compromised rotation key in order to compromise the immediately prior establishment event. The attacker can then rotate in a set of witnesses under its control so that witness compromise is not needed. Notwithstanding this exploit, as the next paragraphs explain, the controller is still protect-

ed against an establishment Dead-attack as long as the original event has had time to propagate as First-seen to any component, such as a watcher the Validator chooses to consult.

Establishment Dead-attack

A successful establishment Dead-attack must first compromise the set of current signing keys for some past stale rotation event, second, create an alternate verifiable version of that stale rotation event, and third, propagate this alternate event to a given validator before the original event has had time to propagate to that validator or any other component the validator may access as First-Seen. The pre-rotation's pre-commitment to the next set of keys means that no other successful establishment Dead-Attack-based exploit is possible. A subsequent rotation event that was not signed with the pre-committed next keys from the prior rotation would not be verifiable. Unlike a non-establishment dead attack, the attacker could rotate in a set of witnesses under its control so that witness compromise is not needed, i.e., the witness pool provides no additional protection. One way to better protect against this exploit is to use partial rotation so that pre-rotated keys are not repurposed as singing keys for interaction events but are first-time, one-time, only-time exposed for signing a rotation. This minimizes the exposure of pre-rotated keys as signing keys and, therefore, minimizes the ability of an attacker to mount an establishment Dead-attack, which requires compromising rotation keys.

To elaborate, compromising a set of keys after the first use, given best practices for key storage and key signing, may be very difficult, but it is still possible. One way to minimize the potential of exploit is to only use rotation keys once using partial rotation. Nonetheless, sometime later, should an attack succeed in compromising a set of stale set of pre-rotated keys and thereby creating an alternate but verifiable event. This may be an eventuality for all non-quantum safe, stale signing and rotation keys.

In any case, a validator or other component may still be protected as long as the original version of the event has had time to propagate as First-seen to that validator or other component (such as witness, watcher, juror, judge) that the validator may access. Therefore, in order to successfully detect duplicity and thereby be protected, any validator needs merely to compare any later copy of the event with any copy of the original event as propagated to any component it may consult. The attacker, therefore, must get ahead of the propagation of a past rotation event. A later surprise quantum attack provides no advantage in this case since the event has already propagated and is already First-seen. The compromised event would be detectable as duplicitous and dropped.

To restate, as already described above, this type of attack looks like what is commonly known as an [24] on a validator. To protect against such an attack, a controller MUST propagate the event sufficiently widely enough that the attacker cannot eclipse all components, such as Watchers, that the validator may consult. The more distant the stale event is in the past the more difficult it becomes to mount a successful eclipse attack because the event would have more time to be universally propagated to the full network of

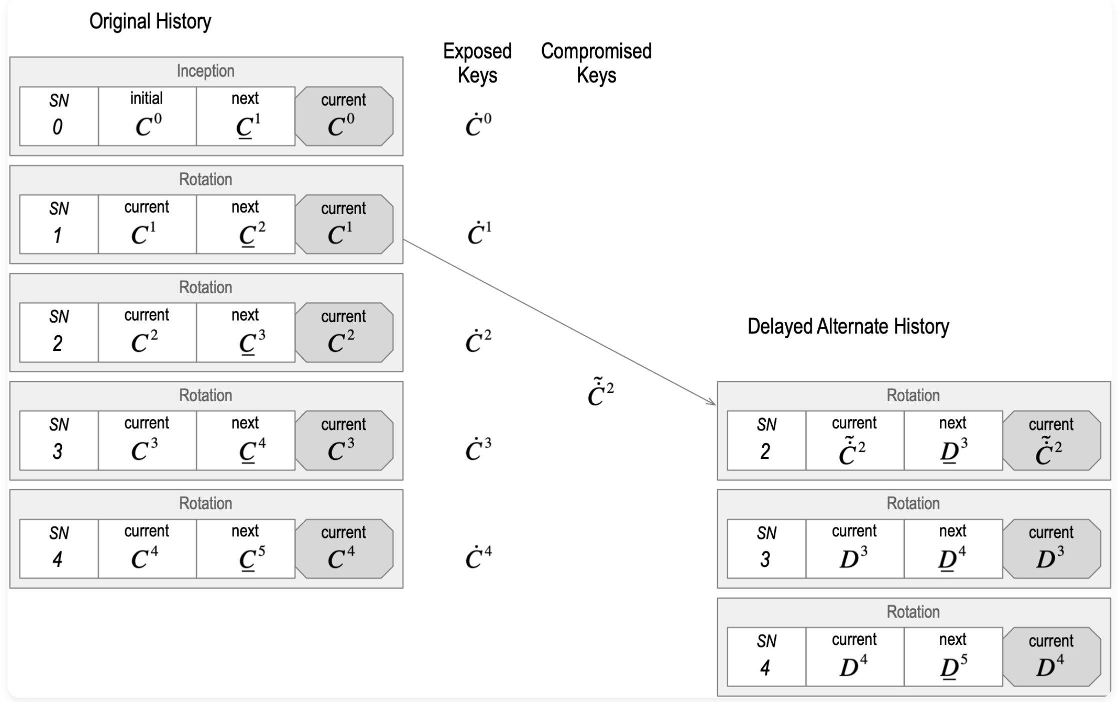
watchers. Otherwise, the Validator would have already First-seen the original event and the compromised event would be dropped i.e., cannot be accepted as First-seen by the Validator. Network propagation times are, at most, seconds and may be as little as milliseconds, which only opens a very short time window of how stale a stale event may be before it is sufficiently protected from any such eclipse attack.

To further elaborate, recall that the original version of the event is the one that first exposes the keys to potential compromise. This may only allow a very narrow window of time for an attacker to get ahead of that event's propagation. In other words, in order for a Dead-Attack to be successful, it must completely avoid detection as duplicitous. To do this, it must either prevent the validator from gaining access to any original copy of the key event history, i.e., an eclipse attack or, equivalently, must first destroy all extant copies of the original key event history accessible to the validator, i.e., some type of deletion attack. This may be very difficult given a sufficiently widespread watcher network.

Moreover, a controller merely needs to receive confirmation via a signed receipt by a validator of its last rotation event to ensure that that validator is protected from future exploitation via deletion Dead Attack. In this case, the controller can replay back to the Validator the Validator's non-reputable signed receipt to recover the Validator from a deletion attack. Likewise, if the controller itself keeps redundant copies of its events, then a deletion attack must completely delete every single copy, otherwise the deletion attack is detectable. A partial deletion attack will always be detectable.

To summarize, an alternate but verifiable version of a rotation event would be detectably inconsistent, i.e., duplicitous with the original version of the event stored in any copy of the original key event history (KEL/KERL). Consequently, any validator (or other component or entity) that has access to the original key event history is protected from harm due to a later successful compromise of the keys of any event already in that history i.e., any form of Dead-attack.

As a special case, to even better protect the initial keypairs in an inception event from a Dead-attack, a controller may coincidently create both the inception event and an immediately following rotation event and then emit them together as one. The initial (original incepting) keypairs may be discarded (including removing all traces from signing infrastructure) after creation but before emission of the coincident events, thereby minimizing the exposure to Dead Attack of these initial keypairs.



Establishment Dead-Attack

Figure: *Establishment Dead-Attack*

Live-Attacks

There are two types of Live-attacks. The first is a compromise of the current signing keys used to sign non-establishment events, such as an interaction event. This is denoted as a non-establishment Live-attack. The second is a compromise of the current pre-rotated keys needed to sign a subsequent establishment event, such as a Rotation event. This is denoted as an establishment Live-attack.

Non-establishment Live-attack

A successful non-establishment Live-attack means that the attacker is able to verifiably sign and propagate a new interaction event. When the interaction event is also protected by a witness pool with a threshold, then the attacker must also compromise a threshold-satisficing number of witnesses, or else the event is not verifiable by any validator. If the witness pool is setup to accept events when merely signed by the controller then the witness pool provides no additional protection from Live-attack. The witness pool merely provides reliable fault tolerant high availability. However, when the members of the witness pool are set up to only accept local (i.e., protected) sourced events from their controller using some unique (per-witness) secondary authentication mechanism, then merely compromising the signing keys is not enough. The attacker must also compromise a threshold-satisficing number of unique secondary authentication factors of the witnesses. A combined primary set of multi-sig controller authentication factors and secondary multi-factor witness authentication factors can make a successful non-establishment Live-at-

tack exploit practically infeasible. Notwithstanding this difficulty, even in the case where a successful non-establishment Live-attack succeeds, control over the identifier can be recovered using a recovery rotation. For more detail see the Annex or Recovery and Reconciliation.

Establishment Live-attack

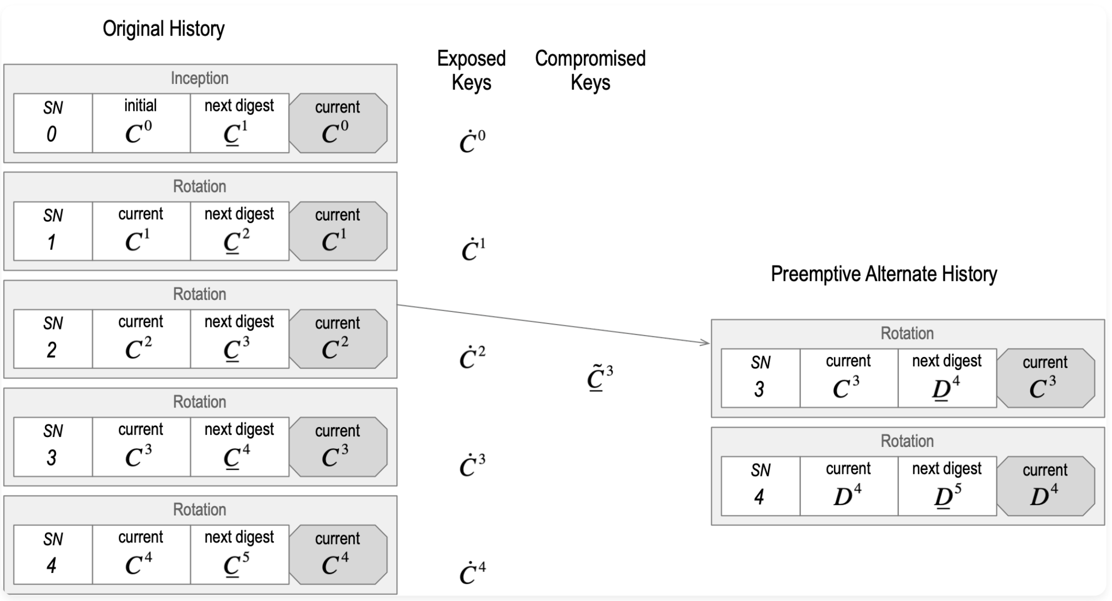
A successful establishment Live-attack means that the attacker somehow compromises the unexposed next (pre-rotated) set of keys from the latest rotation event before that event has been propagated. This means compromise must occur at or before the time of the first use of the keys to sign the establishment event itself. Such a compromise is extremely difficult and the primary reason for pre-rotation is to mitigate the possibility of an establishment Live-attack in the first place. In an establishment live attack, the witness pool provides no additional security because the compromised rotation event can rotate in new witnesses under the control of the attacker. One way to mitigate establishment Live-attack is to use a thresholded multi-sig set of pre-rotated keys.

Notwithstanding any mitigations, assuming a successful compromise of the pre-rotated keys, duplicity detection with or without witness protection may not protect against a resulting establishment live attack. This is because such a Live-attack would be able to create a new verifiable rotation event with next keys and new witnesses under the attacker's control and propagate that event in advance of a new rotation event created by the original controller. Such a successful Live-attack exploit may effectively and irreversibly capture control of the identifier. Moreover, in the case of a successful Live-attack exploit, new rotation events from the original controller would appear as duplicitous to any validator or other component that already received the exploited rotation event and accepted it as the First-seen version of that event. Consequently, protection from establishment Live-attack exploits comes exclusively from the difficulty of compromising a set of pre-rotated keys before or at the time of their first use (exposure).

To elaborate, a successful live exploit must compromise the unexposed next set of private keys from the public-keys declared in the latest rotation. Assuming the private keys remain secret, a compromise must come either by brute force inversion of the one-way digest function protecting the public keys and then by brute force inversion of the one-way scalar multiplication function that generates the public key from the private keys or by a side-channel attack at the first-use of the private keys to sign the rotation event. By construction, no earlier signing side-channel attack is possible. This makes successful Live-attack exploits from such side-channel attacks extremely difficult.

Given the cryptographic strength of the key generation algorithm, a successful brute force live attack may be computationally infeasible. Hiding the unexposed next (pre-rotated) public keys behind cryptographic strength digests provides an additional layer of protection not merely from pre-quantum brute force attacks but also from surprise post-quantum brute force attacks. In this case, a brute force attack would first have to invert

the post-quantum resistant one-way hashing function used to create the digest before it may attempt to invert the one-way public key generation algorithm. Moreover, as computation capability increases, the controller can merely rotate to correspondingly strong quantum-safe cryptographic one-way functions for key generation. This makes brute force live attack computationally infeasible indefinitely. For more detail see the Annex on Cryptographic Strength.



Establishment Live-attack

Figure: *Establishment Live-attack*

Delegated Event Live-attacks

Notwithstanding the foregoing section, delegated events are provided with an additional layer of protection against and an additional means of recovery from establishment Live-attack exploits. As described previously, a delegated event is only valid if the validator finds an anchored delegation seal of the delegated establishment event in the delegator's KEL. This means that notwithstanding a successful compromise of the Delegatee 's current set of pre-rotated keys, the attacker is not able to issue a valid compromised rotation event. The attacker must also issue a delegation seal of the compromised rotation event in the delegator's KEL. This means the attacker must either induce the delegator to issue a seal or must also compromise the delegator's signing keys. This provides an additional layer of protection from establishment Live-attack for delegated events.

Moreover, anytime the sealing (anchoring) event in the delegator's KEL may be superseded by another event, then the delegator and Delegatee may execute a superseding recovery of an establishment event in the Delegatee's KEL and thereby recover from the establishment Live-attack. This is not possible with an establishment Live-attack on a non-delegated event.

Annex A

Cryptographic strength and security

Cryptographic strength

For crypto-systems with perfect-security, the critical design parameter is the number of bits of entropy needed to resist any practical brute force attack. In other words, when a large random or pseudo-random number from a cryptographic strength pseudo-random number generator [CSPRNG] expressed as a string of characters is used as a seed or private key to a cryptosystem with perfect-security, the critical design parameter is determined by the amount of random entropy in that string needed to withstand a brute force attack. Any subsequent cryptographic operations MUST preserve that minimum level of cryptographic strength. In information theory, [10] [ITPS] the entropy of a message or string of characters is measured in bits. Another way of saying this is that the degree of randomness of a string of characters can be measured by the number of bits of entropy in that string. Assuming conventional non-quantum computers, the convention wisdom is that, for systems with information-theoretic or perfect-security, the seed/key needs to have on the order of 128 bits (16 bytes, 32 hex characters) of entropy to practically withstand any brute force attack [12] [11]. A cryptographic quality random or pseudo-random number expressed as a string of characters will have essentially as many bits of entropy as the number of bits in the number. For other crypto-systems such as digital signatures that do not have perfect-security, the size of the seed/key may need to be much larger than 128 bits in order to maintain 128 bits of cryptographic strength.

An N-bit long base-2 random number has 2^N different possible values. Given that no other information is available to an attacker with perfect-security, the attacker may need to try every possible value before finding the correct one. Thus, the number of attempts that the attacker would have to try maybe as much as 2^{N-1}. Given available computing power, one can easily show that 128 bits is a large enough N to make brute force attack computationally infeasible.

The adversary may have access to supercomputers. Current supercomputers can perform on the order of one quadrillion operations per second. Individual CPU cores can only perform about 4 billion operations per second, but a supercomputer will parallelly employ many cores. A quadrillion is approximately 2^{50} = 1,125,899,906,842,624. Suppose somehow an adversary had control over one million (2^{20} = 1,048,576) supercomputers which could be employed in parallel when mounting a brute force attack. The adversary then could try $2^{50} \times 2^{20} = 2^{70}$ values per second (assuming very conservatively that each try took only one operation).

There are about 3600 * 24 * 365 = 313,536,000 = $2^{\log_2 313536000} = 2^{24.91} \sim= 2^{25}$ seconds in a year. Thus, this set of a million super computers could try $2^{50+20+25} = 2^{95}$ values per year. For a 128-bit random number this means that the adversary would need on the or-

der of $2^{128-95} = 2^{33}$ = 8,589,934,592 years to find the right value. This assumes that the value of breaking the cryptosystem is worth the expense of that much computing power. Consequently, a cryptosystem with perfect-security and 128 bits of cryptographic strength is computationally infeasible to break via brute force attack.

Information theoretic security and perfect-security

The highest level of cryptographic security with respect to a cryptographic secret (seed, salt, or private key) is called information-theoretic security. A cryptosystem that has this level of security cannot be broken algorithmically even if the adversary has nearly unlimited computing power including quantum computing. It must be broken by brute force if at all. Brute force means that in order to guarantee success the adversary must search for every combination of key or seed. A special case of information-theoretic security is called perfect-security. Perfect-security means that the ciphertext provides no information about the key. There are two well-known cryptosystems that exhibit perfect-security. The first is a one-time-pad (OTP) or Vernum Cipher; the other is secret splitting, a type of secret sharing that uses the same technique as a one-time-pad.

Post-Quantum Security

Post-quantum or quantum-safe cryptography deals with techniques that maintain their cryptographic strength despite attacks from quantum computers. Because it is currently assumed that practical quantum computers do not yet exist, post-quantum techniques are forward-looking to some future time when they do exist. A one-way function that is post-quantum secure will not be any less secure (resistant to inversion) in the event that practical quantum computers suddenly or unexpectedly become available. One class of post-quantum secure one-way functions are some cryptographic strength hashes. The analysis of D.J. Bernstein with regards the collision resistance of cryptographic one-way hashing functions concludes that quantum computation provides no advantage over non-quantum techniques. Consequently, one way to provide some degree of post-quantum security is to hide cryptographic material behind digests of that material created by such hashing functions. This directly applies to the public keys declared in the pre-rotations. Instead of a pre-rotation making a cryptographic pre-commitment to a public key, it makes a pre-commitment to a digest of that public key. The digest may be verified once the public key is disclosed (unhidden) in a later rotation operation. Because the digest is the output of a one-way hash function, the digest is uniquely strongly bound to the public key. When the unexposed public keys of a pre-rotation are hidden in a digest, the associated private keys are protected from a post-quantum brute force inversion attack on those public keys.

To elaborate, a post-quantum attack that may practically invert the one-way public key generation (ECC scalar multiplication) function using quantum computation must first invert the digest of the public key using non-quantum computation. Pre-quantum cryptographic strength is, therefore, not weakened post-quantum. A surprise quantum capabili-

ty may no longer be a vulnerability. Strong one-way hash functions, such as 256-bit (32-byte) Blake2, Blake3, and SHA3, with 128-bits of pre-quantum strength, maintain that strength post-quantum. Furthermore, hiding the pre-rotation public keys does not impose any additional storage burden on the controller because the controller MUST always be able to reproduce or recover the associated private keys to sign the associated rotation operation. Hidden public keys may be compactly expressed as Base64 encoded qualified public keys digests (hidden) where the digest function is indicated in the derivation code.

KERI Security Properties

Every operation in this protocol is expressed via cryptographically verifiable events. Successful exploitation, therefore, must attack and compromise the availability and/or consistency of events. Security analysis, therefore, is focused on characterizing the nature and timing of these attacks and how well the protocol preserves the availability and consistency of events when subject to attack. Therefore, these potential exploits are described in terms of these properties.

The first property concerns live versus dead event exploits. A live exploit involves attacks on current or recent events. Protection from live exploits is essential to maintaining operational security in the present. Protection from live exploits focuses on providing sufficient availability of current events as well as ensuring their consistency (non-duplicity). A dead exploit, in contrast, involves attacks on past events. Protection from dead exploits is primarily provided by duplicity detection (consistency). One verifiable copy of a KEL (more specifically a KERL) is enough to detect duplicity in any other verifiable but inconsistent copy. Attacks on the availability of past events are relatively easily mitigated by archiving redundant copies. The eventuality of dead exploits of compromised signing keys SHOULD be mitigated because digital signatures may become less secure as computing and cryptographic technology advance over time (quantum or otherwise). Eventually, their keys may become compromised via a direct attack on their cryptographic scheme.

The second property is a direct versus indirect operational mode exploit. The protocol may operate in two basic modes, called direct and indirect. The availability and consistency of attack surfaces are different for the two modes, and hence, the mitigation properties of the protocol are likewise mode-specific.

The third property is a malicious third party versus a malicious controller exploit. In the former, the attack comes from an external malicious attacker, but the controller is honest. In the latter, the controller may also be malicious and, in some ways, may be indistinguishable from a successful malicious third party. The incentive structure for the two exploit types is somewhat different, and this affects the mitigation properties of the protocol. It is helpful in both the design and analysis of protection to consider these two kinds of attacks separately.

The main participants in the protocol are controllers and validators. The other participants, such as witnesses, watchers, jurors, judges, and resolvers, provide support to and may be under the control of either or both of the two main participants.

The analysis of protection against an attack can be further decomposed into three properties of each protection mechanism with respect to an attack: susceptibility to being attacked, vulnerability to harmfulness given an attack, and recoverability given a harmful attack. Security design involves making trade-offs between these three properties of protection mechanisms. Harm from a successful exploit may arise in either or both of the following two cases:

- A controller may suffer harm due to the loss or encumberment of some or all of its control authority such that the malicious entity may produce consistent, verifiable events contrary to the desires of the controller and/or impede the ability of the controller to promulgate new key events.
- A validator may suffer harm due to its acceptance of inconsistent verifiable events produced by a malicious entity (controller and/or third party).

Protection consists of either prevention or mitigation of both of the harm cases. The primary protection mechanisms for the controller include best practice key management techniques for maintaining root control authority, redundant confirmation of events by supporting components, and duplicity detection on the behavior of designated supporting components. The primary protection mechanism for the validator is duplicity detection on the behavior of supporting components.

Validation

Verifier

A verifier is an entity or component that cryptographically verifies an event message's structure and its signature(s). Structure verification includes verifying the event's said, the appearance of fields, and prior event digests. In order to verify signatures, a verifier MUST first determine which set of keys are or were the controlling set for the AID of that event when the event was issued. In other words, a verifier MUST also establish the control authority for the event at issuance. This control establishment requires a copy of the inception event for identifiers that are declared as non-transferable at inception. For identifiers that are declared transferable at inception, this control establishment requires a complete copy of the sequence of key operation events (inception and all rotations) for the identifier up to and including the issued event. Signature verification includes verifying signatures from both current signing and exposed prior rotation (if any) public keys for the event. This includes verifying threshold satisfaction for both current and prior rotation thresholds. Without loss of generality, any reference to the verification of an event or verification of an event's signatures refers to the complete process described above.

Validator

In contrast, a Validator is an entity or component that determines that a given signed event associated with an AID was valid at the time of its issuance. Validation first requires that the event itself is verifiable; that is, it has verifiable structure and signatures from the current controlling key pairs at the time of its issuance. Therefore, a validator MUST first act as a verifier in order to establish the root authoritative set of keys and verify the associated signatures. Once verified, the validator may apply other criteria or constraints to the event in order to determine its validity. This may include witnessing and delegation validation. The final result of validation may be acceptance of the event into the associated KEL. The location of an event in its key event sequence is determined by its sequence number, `sn`. The version of an event at a given location in the key event sequence is different or inconsistent with some other event at the same location if any of its content differs or is inconsistent with that other event.

Duplicity

A duplicitous event is defined as a verified but different version of an event at the same location. The possession of a KEL for any AID enables duplicity detection by a validator for any set of events with respect to that KEL. Indeed, this property of KERI enables duplicity evident processing of events. This forms a basis for evaluating trust in the controller of an AID. A validator can decide to trust or not based on the evidence or lack thereof of duplicity. A validator may choose to trust when there is no evidence of duplicity. A validator SHOULD choose not to trust when there is evidence of duplicity. In some cases, the controller may perform a recovery operation that enables a validator to reconcile that duplicity and allow the validator to once again trust the controller.

Event Types and Classes

In validation, in addition to the version, the type and/or class of event may matter. There are five types of events, inception, `icp`, rotation, `rot`, interaction, `ixn`, delegated inception `dip`, and delegated rotation, `drt`.

There are two main classes of events these are:

- establishment consisting of the types inception, rotation, delegated inception, and delegated rotation.
- non-establishment consisting of the type, interaction.

There is one sub-class, delegated establishment, which consists of the types of delegated inception and delegated rotation.

Validator Roles and Event Locality

Each controller MUST accept its own events into its own copy of its KEL. In this sense, every controller MUST also be a validator for its own AID. Controllers, as validators, play different roles for different types and classes of events. The validation logic for accep-

tance of an event into a given controller's KEL depends on the role that the controller plays for that type or class of event.

The possible roles that a given validator may play for any given event are as follows: controller, witness, delegator, Delegatee, or none of the above. For the sake of clarity, validators that act in different roles with respect to a given event are called the parties to the event. When the context makes it clear, a party that is not one of a controller, witness, delegator, or Delegatee is called simply a validator. Otherwise, the role of the validator is qualified and can be expressed by the roles of controller, witness, validator, delegator, and Delegatee. A party may be referred to as a validator when it does not matter what role a party plays. To clarify, all the parties perform validation, but the validation rules are different for each role. A given controller may also act as a delegator or Delegatee for a given event. A given event for an AID may both delegate other AIDs and be delegated by yet other AIDs. An event is processed differently by each party depending on its respective role or roles.

Events are also processed as either local (protected, trustable) or nonlocal, i.e., remote (unprotected, untrustable). The validator of a local event may assume that the event came from a protected, trustable source, such as the device upon which the validation code is running or from a protected transmission path from some trustable source. The validator of a remote (nonlocal) event MUST assume that the event may have come from a malicious, untrustable source. To elaborate, an event is deemed local when the event is sourced locally on a device under the supervision of the validator or was received via some protected channel using some form of MFA (multi-factor authentication) that the validator trusts. An event is deemed remote (nonlocal) when it is received in an unprotected (untrustable) manner. The purpose of local versus remote is to enable increased security when processing local events where a threshold structure may be imposed, such as a threshold of accountable duplicity for a witness pool.

To elaborate, a witness pool may act as a threshold structure for enhanced security when each witness only accepts local events that are protected by a unique authentication factor in addition to the controller's signatures on the event, thereby making the set of controller signature(s) the primary factor and the set of unique witness authentication factors a secondary thresholded multifactor. In this case, an attacker would have to compromise not merely the controller's set of private key(s) but also the unique second factor on each of a threshold number of witnesses.

Likewise, a delegator may act as a threshold structure for enhanced security when the delegator only accepts local events for delegation that are protected by a unique authentication factor, thereby making the set of controller signature(s) the primary factor, the set of unique witness authentication factors a secondary thresholded multifactor and the delegator's unique authentication factor as a tertiary factor. An attacker, therefore, has to compromise not merely the controller's private key(s) as the primary factor but also the

unique secondary factor on each of a threshold number of witnesses and the unique tertiary factor for the delegator. This layered set of multifactor authentication mechanisms can make exploit practically infeasible.

Validation Rules

The validation rules are as follows:

- Given a local event, the event's controller (non-Delegatee) can sign and accept that event into its copy of that event's KEL. The controller SHOULD then propagate that event with attached controller signatures to the event's witnesses for receipting. Given a remote event, the event's controller (non-Delegatee) SHOULD NOT sign or otherwise accept that event into its copy of that event's KEL. The controller SHOULD NOT propagate that event to the event's witnesses for receipting.
- Given a local event, the event's witness MUST first verify the event's controller signatures before it can sign (witness) and accept that event into its copy of that event's KEL. The witness then SHOULD create a receipt of the event with the attached controller and witness signatures and can then propagate that event to other validators. Given a remote event, the event's witness SHOULD NOT sign or otherwise accept that event into its copy of that event's KEL. The witness SHOULD NOT propagate that event to other validators including witnesses.
- Given a local delegated event, the event's Delegatee can sign and accept that event into its copy of that event's KEL. The Delegatee then SHOULD propagate that event with attached signatures to the event's witnesses for receipting. The Delegatee also SHOULD propagate that event with attached controller signatures and attached witness signatures (if witnessed) to the event's delegator for approval via an anchored seal. Given a remote event, the event's Delegatee SHOULD NOT sign or otherwise accept that event into its copy of that event's KEL. The Delegatee SHOULD NOT propagate that event to the event's witnesses for receipting or to the event's delegator for approval.
- Given a local delegated event, the event's delegator MUST first verify the event's Delegatee signatures and witness signatures (if witnessed) before it can accept that event into its copy of that event's KEL. The delegator can then decide to approve that event by sealing (anchoring the event's seal) the event in the delegator's own KEL. Since the Delegator sealed the event, the delegator then can propagate a receipt of the sealing (anchoring) event to the event's Delegatee and to the event's witnesses (if any). Given a remote delegated event, the event's delegator SHOULD NOT approve (seal) or otherwise accept that event into its copy of that event's KEL. A malicious attacker that compromises the pre-rotated keys of the Delegatee may issue a malicious but verifiable rotation that changes its witness pool in order to by-

pass the local security logic of the witness pool. The approval logic of the delegator SHOULD NOT automatically approve a delegable rotation event unless that event's change to the witness pool is below the witness pool's prior threshold.

- Given a local or remote event, a non-controller, non-witness, non-delegator validator MUST first verify the event's controller signatures, witness signatures (if witnessed), and delegator anchoring seal (if delegated) before it can accept that event into its copy of that event's KEL.
- Given either a local or remote receipt any validator MUST first verify the witness signatures or delegator seals attached to that receipt and then can attach those signatures or seals to its copy of the event.

Superseding Recovery and Reconciliation

First Seen Policy

Once a given Version of an event at a location has been accepted, it is considered "first seen" for that KEL. Once an event has been first seen, it is always seen and can't be unseen. This rule is succinctly expressed as "first seen, always seen, never unseen." This First-seen property enables duplicity detection of different versions of an event. Although an event can never be unseen, in some special cases, it may be superseded by a different version of an event at the same location. Although never explicitly represented in an event message itself, each event belonging to a KEL is also assigned a strictly monotonically increasing integer ordinal called the first-seen number, `fn`, which is stored alongside the event in the KEL database. This allows any copy of a KEL to keep track of the ordering of when each event was first-seen independent of the event's location given by its sequence number, `sn`. Different copies of a KEL may have different first-seen numbers, `fn` for given versions of events at a location, `sn`, but consistent copies of the KEL will have the same version of the event at every location. Events that are superseded are essentially forked. A KEL is essentially a directed acyclic graph (DAG) of events. When an event is superseded, a branch in the DAG is created. There may be only one undisputed path through the DAG called the trunk. All the superseded branches are considered disputed.

Reconciliation

Reconciliation is the process of determining the undisputed path, i.e., the trunk. If every validator of a KEL cannot universally find the same undisputed path (trunk) through a KEL, then the KEL is irreconcilable. Reconciliation happens by applying the superseding validation acceptance (reconciliation) rules to different versions of events that are received for the same location in a KEL. Superseding events provide a universal reconciliation process to enable recovery from key compromises where such key compromises resulted in the first acceptance of compromised events into a KEL. Recovery happens with superseding rotation events that both rotate out the compromised keys and dispute the

events signed by those compromised keys. Because events are signed nonrepudiably, any key compromise is still the responsibility of the controller. That controller still may be held accountable for any harm that resulted from the compromise. However, recovery enables the KEL to be repaired so that future validators of the KEL will not see the compromised events after recovery. The events will only be seen by the validators who first saw the events before recovery.

The superseding validation acceptance rules for events at a given location may involve some combination of the location, version, type, and class of the event as well as the role of the validator, such as controller, witness, delegator, Delegatee, or none of the above.

Superseding Recovery

To supersede an event means that after an event has already been accepted as first seen into a KEL, a different event at the same location (same sequence number) is accepted that supersedes that pre-existing event. The new event becomes part of the trunk (undisputed path), and the old event is the first event in a fork (branch) that includes all the subsequent events to the superseded event. This enables the recovery over the control of a KEI despite events signed by compromised keys. The result of superseded recovery is that the KEL is forked at the `sn` (location) of the superseding event. The compromised events are shunted to the disputed branch, and new events to the KEL are attached to the new end of the trunk formed by the superseding event. To clarify, all events in the superseded branch of the fork still exist but, by virtue of being superseded, are disputed. The set of superseding events in the superseding fork forms the authoritative branch of the KEL, i.e., its trunk or undisputed path. All the already-seen but superseded events in the superseded fork still remain in the KEL. These superceded events may be viewed or replayed in order of their original acceptance because the database stores all accepted events in order of acceptance and denotes this order using the first seen ordinal number, `fn`. The `fn` is not the same as the `sn` (sequence number). Each event accepted into a KEL has a unique `fn` but multiple events due to recovery forks may share the same `sn`. Events with the same `sn` in different copies of the same KEL may have different `fns` because the events may have been first seen or accepted into that KEL in a different order. Notably, an event that may not supersede, according to the rules below, another event at the same location cannot be first seen at all by that KEL. This effectively protects the KEL from later compromise for any downstream recipients of that KEL.

Superseding Rules for Recovery at a given location, SN (sequence number).

A.

A0. Any rotation event may supersede an Interaction event at the same `sn` where that interaction event is not before any other rotation event.

A1. A non-delegated rotation may not supersede another rotation.

A2. An interaction event may not supersede any event.

B. A delegated rotation may supersede the latest-seen delegated rotation at the same `sn` under either of the following conditions:

B1. The superseding rotation's delegating event is later than the superseded rotation's delegating event in the delegator's KEL, i.e., the `sn` of the superseding event's delegation is higher than the `sn` of the superseded event's delegation.

B2. The superseding rotation's delegating event is the exact same delegating event as the superseded rotation's delegating event in the delegator's KEL, and the anchoring seal of the superseding rotation's delegated event appears later in the seal list than the anchoring seal of the superseded rotation's delegated event. i.e., both superseded and superseding event delegations appear in the same delegating event, and the anchoring seal of the superseding rotation's event appears later in the seal list than the anchoring event seal of the superseded rotation's event.

B3. The `sn` of the superseding rotation's delegating event is the same as the `sn` of the superseded rotation's delegating event in the delegator's KEL, and the superseding rotation's delegating event is a rotation, and the superseded rotation's delegating event is an interaction, i.e., the superseding rotation's delegating event is itself a superseding rotation of the superseded rotation's delegating interaction event in its delegator's KEL.

C. IF neither A. nor B. is satisfied, then recursively apply rules A. and B. to the delegating events of those delegating events and so on until either A. or B. is satisfied, or the root KEL of the delegation which MUST be undelegated has been reached.

C1. IF neither A. nor B. is satisfied by the recursive application of C. to each delegator's KEL in turn, i.e., the root KEL of the delegation has been reached without satisfaction, then the superseding rotation is discarded. The terminal case of the recursive application of C. will occur at the root KEL, which by definition MUST be non-delegated therefore either A. or B. MUST be satisfied, or else the superseding rotation MUST be discarded.

The latest-seen delegated rotation constraint in B. means that any earlier delegated rotations can NOT be superseded. This greatly simplifies the validation logic and avoids a potentially infinite regress of forks in the delegated identifier's KEL. However, this means recovery can not happen for any compromise of pre-rotated keys, only the latest-seen. In order to unrecoverably capture control of a delegated identifier, the attacker MUST issue a delegated rotation that rotates to keys under the control of the attacker that the delegator MUST approve and then issue and get approved by the delegator another rotation that follows but does not supersede the compromising rotation. At that point, recovery is no longer possible because the Delegatee would no longer control the private pre-rotated keys needed to sign a recovery rotation as the latest-seen rotation verifiably. Recovery is possible after the first compromised rotation by superseding it but not after the subsequent compromised rotation.

A rotation event at the same location may supersede an interaction. This enables recovery of live exploit of the exposed current set of authoritative keys used to sign non-establishment events via a rotation establishment event to the unexposed next set of authoritative keys. The recovery process forks off a disputed branch from the recovered trunk. This disputed branch has the compromised events, and the main trunk has the recovered events.

KERI's Algorithm for Witness Agreement (KAWA)

Introduction

A controller may provide a highly available promulgation service for its events via a set or pool of N designated witnesses. This Witness pool may also provide enhanced security for the controller over its events. Even though the witnesses are explicitly designated by the controller, they may or may not be under the control of the controller. The designation is a cryptographic commitment to the witnesses via a verifiable statement included in an Establishment event. The purpose of the witness set is to better protect the service from faults including Byzantine faults. Thus, the service employs a type of Byzantine Fault Tolerant (BFT) algorithm called KERI's Algorithm for Witness Agreement (KAWA). The primary purpose of KAWA is to protect the controller's ability to promulgate the authoritative copy of its key event history despite external attacks. This includes maintaining a sufficient degree of availability such that any Validator may obtain an authoritative copy on demand.

The critical insight is that because the controller is the sole source of truth for the creation of any and all key events, it alone, is sufficient to order its own key events. Indeed, a key event history does not need to provide double spend proofing of an account balance, merely consistency. Key events, by and large, are idempotent authorization operations as opposed to non-idempotent account balance decrement or increment operations. Total or global ordering may be critical for non-idempotency, whereas local ordering may be sufficient for idempotency, especially to merely prove the consistency of those operations. The implication of these insights is that fault tolerance may be provided with a single-phase agreement by the set of witnesses instead of a much more complex multi-phase commit among a pool of replicants or other total ordering agreement process as is used by popular BFT algorithms. Indeed, the security guarantees from KAWA may approach that of other BFT algorithms but without their scalability, cost, throughput, or latency limitations. If those other algorithms may be deemed sufficiently secure, then so may be KAWA. Moreover, because the controller is the sole source of truth for key events, a validator may hold that controller (whether trusted or not) accountable for those key events. As a result, the algorithm is designed to enable a controller to provide itself with any degree of protection it deems necessary given this accountability.

Advantages

The reliance on a designated set of witnesses provides several advantages. The first is that the identifier's trust basis is not locked to any given witness or set of witnesses but may be transferred at the controller's choosing. This provides portability. The second is that the number and composition of witnesses is also at the controller's choosing. The controller may change this in order to make trade-offs between performance, scalability, and security. This provides flexibility and adaptability. Thirdly, the witnesses need not provide much more than verification and logging. This means that even highly cost or performance constrained applications may take advantage of this approach.

Likewise, given any guarantees of accountability the controller may declare, a validator may provide itself with any degree of protection it deems necessary by designating a set of observers (watchers, jurors, and judges). Specifically, a validator may be protected by maintaining a copy of the key event history as first seen (received) by the validator or any other component trusted by the validator (watcher, juror, judge). This copy may be used to detect any alternate inconsistent (duplicitous) copies of the key event history. The validator then may choose how to best respond in the event of a detected duplicitous copy to protect itself from harm. A special case is a malicious controller that intentionally produces alternate key event histories. Importantly, observer components that maintain copies of the key event history such as watchers, jurors, and judges, may be under the control of validators not controllers. As a result, a malicious alternate (duplicitous) event history may be eminently detectable by any validator. This is called ambient duplicity detection (which stems from ambient verifiability). In this case, a validator may still be protected because it may still hold such a malicious controller accountable given a duplicitous copy (trust or not trust). It is at the validator's discretion whether or not to treat its original copy as the authoritative one with respect to any other copy and thereby continue trusting or not that original copy. A malicious controller may not therefore substitute later with impunity any alternate copy it may produce. Furthermore, as discussed above, a malicious controller that creates an alternative event history imperils any value it may wish to preserve in the associated identifier. The alternative event history is potentially completely self-destructive with respect to the identifier. A malicious controller producing a detectably duplicitous event history is tantamount to a detectable total exploit of its authoritative keys and the keys of its witness set. This is analogous to a total but detectable exploit of any BFT ledger such as a detectable 51% attack on a proof-of-work ledger. A detectable total exploit destroys any value in that ledger after the point of exploit.

To restate, a controller may designate its witness set in such a way as to provide any arbitrary degree of protection from external exploit. Nonetheless in the event of such an exploit a validator may choose either to hold that controller accountable as duplicitous and therefore stop trusting the identifier or to treat the validator's copy of the key event history as authoritative (ignoring the exploited copy) and therefore continue trusting the identifier. This dependence on the validator's choice in the event of detected duplicity both imperils any potential malicious controller and protects the validator.

KERI's KAWA or the algorithm is run by the controller of an identifier in concert with a set of N witnesses designated by the controller to provide as a service the key event history of that identifier via a KERL in a highly available and fault-tolerant manner. One motivation for using key event logs is that the operation of redundant immutable (deletion proof) event logs may be parallelizable and hence highly scalable. A KERL is an immutable event log that is made deletion proof by virtue of it being provided by the set of witnesses of which only a subset of F witnesses may at any time be faulty. In addition to designating the witness set, the controller also designates a threshold number, M, of witnesses for accountability. To clarify, the controller accepts accountability for an event when any subset M of the N witnesses confirms that event. The threshold M indicates the minimum number of confirming witnesses the controller deems sufficient given some number F of potentially faulty witnesses. The objective of the service is to provide a Verifiable KERL to any validator on demand. Unlike direct mode where a validator may be viewed as an implicit witness, with indirect mode, a validator may not be one of the N explicitly designated witnesses that provide the service.

Witness Designation

The controller designates both the witness tally number and the initial set of witnesses in the Inception event configuration. The purpose of the tally is to provide a threshold of accountability for the number of witnesses confirming an event. Subsequent rotation operations may amend the set of witnesses and change the tally number. This enables the controller to replace faulty witnesses and/or change the threshold of accountability of the witness set. When a rotation amends the witnesses it includes the new tally, the set of pruned (removed) witnesses and the set of newly grafted (added) witnesses.

Witnessing Policy

In this approach, the controller of a given identifier creates and disseminates associated key event messages to the set of N witnesses. Each witness verifies the signatures, content, and consistency of each key event it receives. When a verified key event is also the first seen version of that event the witness has received, then it witnesses that event by signing the event message to create a receipt, storing the receipt in its log (KERL), and returning the receipt as an acknowledgment to the controller. Depending on its dissemination policy, a witness may also send its receipt to other witnesses. This might be with a broadcast or gossip protocol or not at all.

In general, the witnessing policy is that the first seen version of an event always wins; that is, the first verified version is witnessed (signed, stored, acknowledged, and maybe disseminated), and all other versions are discarded. The exception to this general rule is that a rotation event may provide a superseding recovery. The recovery process may fork off a branch from the recovered trunk. This disputed branch has the disputed exploited

events, and the main trunk has the recovered events. The operational mode and the threshold of accountable duplicity determine which events in the disputed branch are accountable to the controller.

Later messages or receipts from other witnesses may not change any existing entry in the log (the log is append-only, i.e., immutable) unless they are correctly reconcilable superseding events. Each witness also adds to its log any verified signatures from consistent receipts it receives from other witnesses. A consistent receipt is a receipt for the same version of the event already in its log at a location. Excepting superseding recovery, inconsistent receipts, i.e., for different event versions at the same location, are discarded (not kept in the log). However, as an option, a controller may choose to run a juror (in concert with a witness) that keeps a duplicitous event log (DEL) of the inconsistent or duplicitous receipts that a witness receives. To clarify, a witness' KERL is by construction, an immutable log. This log includes the events with attached verified signatures, which are the receipts from the controller, the witness, and other witnesses.

Initial dissemination of receipts to the N witnesses by the controller may be implemented extremely efficiently with respect to network bandwidth using a round-robin protocol of exchanges between the controller and each of the witnesses in turn. Each time the controller connects to a witness to send new events and collect the new event receipts, the Controller also sends the receipts it has received so far from other witnesses. This round-robin protocol may require the controller to perform at most two passes through the entire set of witnesses in order to fully disseminate a receipt from each witness to every other witness for a given event. This means that at most 2·N acknowledged exchanges are needed for each event to create a fully witnessed KERL at every witness and controller. Network load, therefore, scales linearly with the number of witnesses.

When network bandwidth is less constrained, a gossip protocol might provide full dissemination with lower latency than a round-robin protocol but with higher bandwidth usage. Gossip protocols scale with N · log(N) (where N is the number of witnesses) instead of 2·N. A directed acyclic graph or other data structure can be used to determine what needs to be gossiped.

Immunity and Availability

It can be shown that for any set of N witnesses, there is a threshold M < N that guarantees that at most one sufficient agreement occurs or none at all, despite a dishonest controller — but where at most F* = N-M of the witnesses are potentially unavailable and at most F < M is duplicitous. This guarantee means that the agreement is deemed immune (from failure due to faulty F or F*). A Controller MAY choose to use the KAWA algorthm to achieve immunity.

Given the immune constraint is satisfied, the service may not produce multiple divergent but proper KERL. In order to be deemed proper, an agreement must have been verified as consistent with all prior events by every non-faulty witness who is a party to that

agreement. Thus, any user of the service, be it a validator, watcher, juror, or judge, will be able to obtain either a proper event agreement on demand from some witness or none at all. Any non-faulty witness with a proper agreement will keep that agreement in its KERL and provide it on demand. Consequently, the availability of a proper event at a witness is tantamount to the availability of a proper log (KERL) of all prior events consistent with that event at that witness, and thereby, high availability of the service is assured.

Security Properties

The continuing promulgation of key events assumes a sufficiently responsive controller. Lack of responsiveness is primarily a threat to the controller, not a validator. Consequently, providing sufficient controller responsiveness is the controller's responsibility, not of KAWA. In contrast, a responsive but dishonest (or compromised) controller may pose a live threat to a validator with respect to new events never before seen by the validator. The validation process MUST provide means for the validator to protect itself from such threats. When the controller is responsive but dishonest, the Controller may create inconsistent versions of an event that are first seen by different subsets of its witnesses. In the case where only F of the witnesses is faulty despite a dishonest controller, the validator may protect itself by requiring a large enough sufficient agreement or threshold of accountable duplicity, M, that guarantees that either only one satisfying agreement or none at all, e.g., makes the service immune. To restate, the validator may select its M to ensure that the service is immune such that the service will either provide one and only one KERL or none at all. This protects the validator.

A greater threat to a validator may be that of a dishonest controller that may collude with its witnesses to promulgate alternative (divergent) event version agreements, each with sufficient agreement. But this would violate the assumption of at most F faulty witnesses. In this case, the witness consensus process, i.e., the KAWA algorithm, may not protect the validator. Protection MUST come from some other process under the validator's control. In this case, a validator may protect itself with duplicity detection via a set of observers (validators, watchers, jurors, judges). In such a case, in order to undetectably promulgate alternate but sufficiently accountable event version agreements, a dishonest controller with dishonest witnesses MUST prevent any validator from communicating with any other observer who may have seen any alternate event version agreement. This attack may be made practically unfeasible given a large and diverse enough set of observers. Indeed, once duplicity is detected, that identifier loses all its value to any detecting validator. This imperils any dishonest controller who attempts such an attack.

The final threat is the threat of dead exploit where, sometime in the future, the exposed key pairs used to sign past events in a KERL may be compromised. The compromised keys may then be used to create an alternate or divergent verifiable event history. Recall, however, that a proper KERL enables validation of the controlling keys of the associated identifier over the time frame of the events in the log. Once produced, a proper KERL

may be provided by any observer (validator, watcher, juror, or judge) that has retained a copy of it not merely the witnesses. Subsequent compromise of a controller's keys and a compromise of witnesses may not invalidate any of the events in a pre-existent proper KERL.

Therefore, in order to fool a validator into accepting an erroneous or compromised divergent Key event history, a successful exploiter must forge a proper KERL but with a different sequence of key events. To do this the exploiter must not only exploit the controller's signing keys that were authoritative at some event but also exploit M of the N designated witnesses at that event as well. The exploiter must also prevent that validator from accessing any other but alternate proper KERL from any other observer (validator, watcher, juror, judger) that may have a copy as a check against such an attack. The combination of these tasks makes such an exploit extremely difficult to achieve.

Consequently, even in the extreme case that sometime in the future, a complete and total dead exploit of the controller keys and at least M of the witnesses occurs such that they forge a seemingly proper but divergent KERL, any prior copy of a proper KERL will enable detection and proof of accountable duplicity of that dead exploit. In this case, the validator may choose to use the prior copy from some set of jurors it trusts to determine which of the divergent KERLs is authoritative. This is similar to how certificate transparency works. In order for such a dead attack to succeed, the attacker must prevent a targeted validator from accessing any other copies of an alternate KERL.

The idea of ambient verifiability mentioned above comes from the fact that the original KERL may be distributed among any number of watchers from whom a validator may obtain a copy. At some point, the degree of accessibility to an original copy becomes essentially ubiquitous, at which point verifiability may be considered ambient. Given ambient verifiability, then, duplicity detection becomes likewise ambient.

To elaborate, a successful dead attack requires the isolation of a validator from ambient sources of the KERL. In general, isolation from ambient sources may be prohibitively expensive. Consequently, ambient verifiability provides asymmetry between the attacker and the defender in favor of the defender. Indeed, the end goal of KERI is to achieve ambient security in the sense that nearly anyone, anywhere, at any time, can become a verifiable controller of a verifiable identity that is protected by ambient verifiability and hence duplicity detection of the associated KERL.

Furthermore, any mutual interaction events between a validator and controller may provide proof of priority. In a mutual interaction, the validator includes a copy or digest of an interaction event sourced by the controller in an event sourced by the validator. A total compromise of the controller and all witnesses would not be able to forge the validator's signature on the mutual interaction event. Thus, the existence of any mutual interaction events may then be used to prove priority even in the extremely unlikely case of a complete and total dead exploit of a controller and all of its witnesses.

Alternatively, in the case of a complete and total dead exploit, the validator and controller may jointly agree to use some other, more formal mechanism to resolve the priority of divergent KERLs. This may be the median of the astronomical time of the original reception of a receipt by a mutually trusted set of observers. This may be through the use of anchor transactions on a distributed consensus ledger. This later approach would only require minimal use of a distributed consensus ledger in order to resolve the most extreme and unlikely case of total dead exploit.

Finally, however unlikely, subsequent improvements in cryptographic attack mechanisms such as quantum computing may enable, at some future time, complete compromise of all exposed key pairs. One solution would be for the market to operate a trusted set of jurors that archive KERLs just in case of some such future total compromise. These trusted jurors may secure their archives with post-quantum cryptography. Thus, any post-quantum attack may be detectable merely by appeal to one or more of these archives.

Working Examples Setup

The code to generate the working examples in this specification is provided via unit tests found in tests/spec/keri in the keripy library.

A brief explanation of the setup code is provided here to help implementers who wish to reproduce the examples from scratch.

AIDs

The examples require an Issuer AID. This AID is created in accordance with the KERI protocol. This requires creating digital signing key-pairs whose public keys are used in an inception event to create the AID. The keripy library has a utility class Salter (found in keri.core.signing.Salter) that facilitates the creation of signing key pairs. For the examples, any needed signing key pairs can be recreated using a known non-random `Salt`. The known non-random salt is merely for reproducibility. In a real-world application, the salt should be a high entropy secret. The salt used for the examples is the python byte string as follows:

```
b'kerispecworkexam'
```

Using a Salter instance, a set of key pairs may be created. These keypairs are instantiated as Signer class instances. The Signer class can be found in keri.core.signing.Signer. The creation code is as follows:

```
salt = b'acdcspecworkexam'
salter = Salter(raw=salt)
signers = salter.signers(count=8, transferable=True, temp=True)
```

The `Salter.signers()` method creates a count number of Signer instances and returns them in a list. Each Signer holds a key pair. The private key seed for each key pair is created using Argon2 to stretch a deterministic path that is based on the salt and a path that is the hex representation of the count offset for each Signer. For the zeroth signer this is as follows:

```
import pysodium
seed = pysodium.crypto_pwhash(outlen=32, passwd='0', salt=b'acdcs
pecworkexam', opslimit=1,
                                memlimit=8192, alg=pysodium.crypt
o_pwhash_ALG_ARGON2ID13)
```

The seed becomes the private signing key for the keypair. The public verification key is generated using Ed25519 as follows:

```
verkey, sigkey = pysodium.crypto_sign_seed_keypair(seed)
```

The verkey is used as the raw input to a Verfer (verifier) Class instance (found in keri.-core.coring.Verfer) as follows:

```
verfer = Verfer(raw=verkey, code=MtrDex.Ed25519)
```

From this, the initial signing public verification key in CESR encoded qualified Base64 `Text` domain representation is as follows:

```
'DA8-J0EW88RMYqtUHQDqT4q2YH2iBFlW8HobHKV74yi_'
```

In order to create the Issuer AID, two other signing key pairs are needed. One other is the “next” rotating key pair. The signer at index 1 is used for this. From this, the next rotating public verification key in CESR encoded qualified Base64 `Text` domain representation is as follows:

```
'DLe4uewytqfqa4NB4AntNKBZ61I0TYcgMz-FSz1V9qeM'
```

Another one is a witness key pair. Witness AIDs are usually non-transferable and use a non-transferable CESR encoding. This is produced by setting the transferable parameter to `False`. The associated key pairs may be generated as follows:

```
walt = b'acdcspecworkwits'
walter = Salter(raw=walt)
wigners = walter.signers(count=4, transferable=False, temp=True)
```

The signer at index 0 (wigners[0]) is used for this. From this, the witness AID, (which is also its signing public verification key) in CESR encoded qualified Base64 `Text` domain representation is as follows:

```
'BKRaC6UsijUY1FRjExoAMc8WOHBDIfIKYnOlxWH8eOe8'
```

From these three Cryptographic primitives, we can create a Python dictionary with all the data needed to generate the inception event for the Issuer as follows:

```
 sad = \
 {
        'v': 'KERICAACAAJSONAAFb.',
        't': 'icp',
        'd': 'ECmiMVHTfZIjhA_rovnfx73T3G_FJzIQtzDn1meBVLAz',
        'i': 'ECmiMVHTfZIjhA_rovnfx73T3G_FJzIQtzDn1meBVLAz',
        's': '0',
        'kt': '1',
        'k': ['DA8-J0EW88RMYqtUHQDqT4q2YH2iBFlW8HobHKV74yi_'],
        'nt': '1',
        'n': ['DLe4uewytqfqa4NB4AntNKBZ61I0TYcgMz-FSz1V9qeM'],
        'bt': '1',
        'b': ['BKRaC6UsijUY1FRjExoAMc8WOHBDIfIKYnOlxWH8eOe8'],
        'c': [],
        'a': []
}
```

The JSON examples use a JSON serialization of the inception event to generate the Issuer AID. In Python, this is performed as follows:

```
import json

raw = json.dumps(sad, separators=(",", ":"), ensure_ascii=False).
encode()
```

This results in the following raw JSON for the inception event.


```
(b'{"v":"KERICAACAAJSONAAFb.","t":"icp","d":"ECmiMVHTfZIjhA_rovnf
x73T3G_FJzIQtz'
b'Dn1meBVLAz","i":"ECmiMVHTfZIjhA_rovnfx73T3G_FJzIQtzDn1meBVLA
z","s":"0","kt":'
b'"1","k":["DA8-J0EW88RMYqtUHQDqT4q2YH2iBFlW8HobHKV74yi_"],"n
t":"1","n":["DLe4'
b'uewytqfqa4NB4AntNKBZ61I0TYcgMz-FSz1V9qeM"],"bt":"1","b":["BKRaC
6UsijUY1FRjEx'
b'oAMc8WOHBDIfIKYnOlxWH8eOe8"],"c":[],"a":[]}')
```

The calculation of the SAIDed fields in the inception event requires knowledge of the SAID protocol for generating SAIDs on the serialization of the associated field map. A utility function for generating an inception event may be found in keri.core.eventing.incept. The raw above was generated as follows:

```
from collections import namedtuple

issuerSigner = signers[0]
issuerVerKey = issuerSigner.verfer.qb64  # issuer's public verifi
cation key
assert issuerVerKey == 'DA8-J0EW88RMYqtUHQDqT4q2YH2iBFlW8HobHKV74
yi_'

issuerRotSigner = signers[1]
issuerRotVerKey = issuerRotSigner.verfer.qb64  # issuer's public
verification key
assert issuerRotVerKey == 'DLe4uewytqfqa4NB4AntNKBZ61I0TYcgMz-FSz
1V9qeM' # use in example

issuerWitSigner = wigners[0]
issuerWitVerKey = issuerWitSigner.verfer.qb64  # issuer's public
verification key
assert issuerWitVerKey == 'BKRaC6UsijUY1FRjExoAMc8WOHBDIfIKYnOlxW
H8eOe8' # use in example

Versionage = namedtuple("Versionage", "major minor")
Vrsn_2_0 = Versionage(major=2, minor=0)  # KERI Protocol Version
Specific

assert MtrDex.Blake3_256 == 'E'

keys = [issuerVerKey]  # initial signing keys
nkeys = [issuerRotVerKey]  # next (rotation) keys
wits = [issuerWitVerKey]  # witness aids (same as public verkey)
serder = incept(keys, code='E', ndigs=nkeys, wits=wits, version=V
rsn_2_0, kind='JSON')
```

The resultant Issuer AID, namely:

```
"ECmiMVHTfZIjhA_rovnfx73T3G_FJzIQtzDn1meBVLAz"
```

It can be used in an example. This AID may be given the user friendly alias `amy` has in Amy's AID. Likewise, the same process can be followed to created other AID's used in examples.

UUIDs

Many of the examples include UUID, `u` fields with salty nonce values. For ease of reproducibility, deterministic UUIDs are used. These may be generated with the following Python code snippet:

```
from keri.core import Noncer

raws = [b'kerispecworkraw' + b'%0x'%(i, ) for i in range(16)]
uuids = [Noncer(raw=raw).qb64 for raw in raws]
assert uuids == \
[
    '0ABrZXJpc3BlY3dvcmtyYXcw',
    '0ABrZXJpc3BlY3dvcmtyYXcx',
    '0ABrZXJpc3BlY3dvcmtyYXcy',
    '0ABrZXJpc3BlY3dvcmtyYXcz',
    '0ABrZXJpc3BlY3dvcmtyYXc0',
    '0ABrZXJpc3BlY3dvcmtyYXc1',
    '0ABrZXJpc3BlY3dvcmtyYXc2',
    '0ABrZXJpc3BlY3dvcmtyYXc3',
    '0ABrZXJpc3BlY3dvcmtyYXc4',
    '0ABrZXJpc3BlY3dvcmtyYXc5',
    '0ABrZXJpc3BlY3dvcmtyYXdh',
    '0ABrZXJpc3BlY3dvcmtyYXdi',
    '0ABrZXJpc3BlY3dvcmtyYXdj',
    '0ABrZXJpc3BlY3dvcmtyYXdk',
    '0ABrZXJpc3BlY3dvcmtyYXdl',
    '0ABrZXJpc3BlY3dvcmtyYXdm'
]
```

The Noncer class may be found in keri.core.coring.Noncer. Essentially, a Noncer instance can encode a byte string as a salty nonce in CESR format.

Native CESR Encodings of KERI Messages

A native CESR encoding of the field map of a KERI message body is represented using pure CESR instead of JSON, CBOR, or MGPK. Because the top-level fields in every KERI message body are fixed and each value in CESR is self-describing and self-framing, there is no need to provide labels at the top level, only the field values in a fixed order. In the following tables, for comparison and clarity, the first column provides the equivalent field label as would be used in JSON, CBOR, or MGPK; the second column provides the field value format; and the third column a short description. For field values that are primitives, an example primitive may be provided as the value. To restate, no top-level labels appear in an actual serialized native CESR message body, just the concatenated field values either as primitives or groups of primitives with the appropriate prepended CESR group codes. The order of appearance of fields as values is strict.

Nested field maps may appear as values of top-level fields. For example, a seal may be expressed as a field map. In that case, the CESR count code for a generic field map is used.

Similarly, lists may appear as values of top-level fields. For example, the current signing keys are expressed as a key list. In that case, the CESR count code for a generic list is used.

CESR Field Encodings

Some field values in KERI messages that include non-Base64 characters have custom CESR Text domain encodings (Base64). These encodings are more compact than the case given the direct conversion of a binary string with non-Base64 characters into Base64.

Protocol and Genus Version

In a CESR-encoded message, the count code for the message includes the message size. This means that the version field in the messages does not need to include the size. Likewise the serialization kind of CESR is determined by the first tritet of the first character of a given message, so the version does not need to include the serialization kind. The only information that MUST be encoded in a native CESR message's version field is the protocol, the protocol version and the genus version of the associated code table. This assumes that for each protocol the genus is fixed so that only the genus version is required not the genus itself given the protocol.

The protocol type uses four characters and MUST be `KERI` for a normative KERI protocol.

The protocol version uses three Base64 characters. The first one provides the major version. The second two provide the minor version. For example, `AAB` represents a major version of `0` and a minor version of decimal `1`. In dotted notation, this would be `0.1`. This provides for a total of 64 major versions and 4096 minor versions for each major version.

The CESR genus version uses three Base64 characters. The first one provides the major version. The second two provide the minor version. For example, `AAB` represents a major version of `0` and a minor version of decimal `1`. In dotted notation, this would be `0.1`. This provides for a total of 64 major versions and 4096 minor versions for each major version.

The encoded protocol type, protocol version, and genus version consume 10 Base64 characters. The CESR primitive code for such a 10-character primitive is `0O`. An example KERI protocol type/protocolversion/genusversion field value for protocol version 2.0 and genus version 2.0 is as follows:

`0OKERICAACAA`

DateTime

As described above, the datetime, `dt` field value, if any, MUST be the ISO-8601 datetime string with microseconds and UTC offset as per IETF RFC-3339. An example datetime string in this format is as follows:

`2020-08-22T17:50:09.988921+00:00`

Because this field value format employs non-Base64 characters, direct conversion to Base64 would increase the size of the value. Instead, the non-Base64 characters are converted to unique Base-64 characters using the following conversion table:

Non-Base64	Base64	Description
`:`	`c`	colon
`.`	`d`	dot
`+`	`p`	plus

Using this conversion, the base CESR encoding of the DateTime example above becomes:

`2020-08-22T17c50c09d988921p00c00`

The CESR code for DateTime is prepended to this string to produce the fully qualified CESR encoding. This encodes the datetime compactly into Base64 without doing a direct Base64 conversion of a binary string.

Threshold

As described above, the fractionally weighted threshold field value may be represented as a list of lists of fractional weights. Those fractional weights may be simple or complex. A complex fractional weight is a single-element field map whose key is a fractional weight and whose value is a list of fraction weights. When serialized in JSON, this field value format employs non-Base64 characters. An example fractionally weighted threshold in JSON is as follows:

```
[[{"1/2": ["1/2", "1/2", "1/2"]}, "1/2", {"1/2": ["1", "1"]}],
["1/2", {"1/2": ["1", "1"]}]]
```

Direct conversion to Base64 would increase the size of the value. Instead, the block delimited expressions of lists, `[]` and field maps, `{:}` are converted to infix operators in Base64. Likewise, the non-Base64 separators for fractions and list elements are converted to infix operators as separators in Base64. The precedence order of the infix operators enables round-trip conversion between the block-delimited non-Base64 expression and the infix Base64 expression. Noteworthy is that the infix operator expression does

not allow recursive nesting. This is not a problem because only one layer of nesting is supported. The following table provides the infix operators in order of priority. Higher in the table is a higher priority.

Non-Base64 Operation	Base64 Infix Operator	Description
`/`	`s`	slash
`{:[,]}`	`k`	key of map
`{:[,]}`	`v`	nested weight list element of map value
`[,]`	`c`	simple weight list element
`[[],[]]`	`a`	ANDed weight list]

Using this infix conversion, the CESR encoding of the JSON fractionally weighted threshold example above becomes:

`1s2k1s2v1s2v1s2c1s2c1s2k1v1a1s21s2k1v1`

Because thresholds are variable length, the appropriate fully qualified CESR code for Base64 only variable-length strings is prepended to the threshold. This encodes the threshold compactly into Base64 without doing a direct Base64 conversion of a binary string.

Route or Return Route

As described above, the value of a Route or Return Route field is a slash, `/` delimited path. Because the slash, `/`, is a non-Base64 character, direct conversion of the route string as binary to Base64 would increase the size of the value. Instead, slashes, `/` may be converted to dashes, `-`. This conversion is only applicable to a route that does not otherwise have any dash, `-` characters, or non-Base64 characters besides the slash, `/.` In that case, the route MUST be converted as binary to Base64. An example route is as follows:

`/ipex/offer`

Converting the slashes to dashes gives:

`-ipex-offer`

In the case where a route may be converted to Base64 characters by merely subsituting dashes, `-` for slashes, `/` then a fully qualified CESR Text domain representation may be created by prepending the appropriate CESR code for variable length Base64 strings. This encodes the route compactly into Base64 without doing a direct Base64 conversion

of a binary string. Otherwise, the route is treated as a variable-length binary string and is converted to Base64. In that case, the appropriate CESR code for variable-length binary strings is prepended to the converted route to provide the Text Domain encoding.

Key Event Messages

These have the following packet types: `[icp, rot, ixn, dip, drt]`.

The examples in this annex are identical to the examples provided in the body of the specification above except that the serialization kind is CESR instead of JSON. This changes any fields that are SAIDive, i.e. are derived from digest of the message body using the SAID protocol. All the keys and UUIDs are the same as the examples above. The other field value that changes is the version string that appears in the Python dict expression of a message body. This is because the size and serialization kind both appear in the version string and both of these are affected by the serialization kind.

Inception `icp`

Field order by label: `v`, `t`, `d`, `i`, `s`, `kt`, `k`, `nt`, `n`, `bt`, `b`, `c`, `a`.

Message body as a Python dict.

```
{
        'v': 'KERICAACAACESRAAJM.',
        't': 'icp',
        'd': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
        'i': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
        's': '0',
        'kt': '2',
        'k':
        [
            'DBFiIgoCOpJ_zW_OO0GdffhHfEvJWb1HxpDx95bFvufu',
            'DG-YwInLUxzVDD5z8SqZmS2FppXSB-ZX_f2bJC_ZnsM5',
            'DGIAk2jkC3xuLIe-DI9rcA0naevtZiKuU9wz91L_qBAV'
        ],
        'nt': '2',
        'n':
        [
            'ELeFYMmuJb0hevKjhv97joA5bTfuA8E697cMzi8eoaZB',
            'ENY9GYShOjeh7qZUpIipKRHgrWcoR2WkJ7Wgj4wZx1YT',
            'EGyJ7y3TlewCW97dgBN-4pckhCqsni-zHNZ_G8zVerPG'
        ],
        'bt': '3',
        'b':
        [
            'BGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3B',
            'BJfueFAYc7N_V-zmDEn2SPCoVFx3H20alWsNZKgsS1vt',
            'BAPv2MnoiCsgOnklmFyfU07QDK_93NeH9iKfOy8V22aH',
            'BA4PSatfQMw1lYhQoZkSSvOCrE0Sdw1hmmniDL-yDtrB'
```

```
        ],
        'c': ['DID'],
        'a': []
    }
```

CESR serialization as a Python byte string.


```
(b'-FCS0OKERICAACAAXicpEDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frT
YsEDZOA3y_b_0L'
b'G4_cfpKTbWU-_3eeYNM0w9iTkT7frTYsMAAAMAAC-JAhDBFiIgoCOpJ_zW_OO0G
dffhHfEvJWb1H'
b'xpDx95bFvufuDG-YwInLUxzVDD5z8SqZmS2FppXSB-ZX_f2bJC_ZnsM5DGIAk2j
kC3xuLIe-DI9r'
b'cA0naevtZiKuU9wz91L_qBAVMAAC-JAhELeFYMmuJb0hevKjhv97joA5bTfuA8E
697cMzi8eoaZB'
b'ENY9GYShOjeh7qZUpIipKRHgrWcoR2WkJ7Wgj4wZx1YTEGyJ7y3TlewCW97dgBN
-4pckhCqsni-z'
b'HNZ_G8zVerPGMAAD-JAsBGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3
BBJfueFAYc7N_'
b'V-zmDEn2SPCoVFx3H20alWsNZKgsS1vtBAPv2MnoiCsgOnklmFyfU07QDK_93Ne
H9iKfOy8V22aH'
b'BA4PSatfQMw1lYhQoZkSSvOCrE0Sdw1hmmniDL-yDtrB-JABXDID-JAA')
```


Interaction `ixn`

Field order by label: `v`, `t`, `d`, `i`, `s`, `p`, `a`.

Message body as a Python dict.

```
{
    'v': 'KERICAACAACESRAAEA.',
    't': 'ixn',
    'd': 'EDmgVuwPOXDjIW3reg4_k8SeJoQEKJKP24fGzeMV4uKD',
    'i': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
    's': '1',
    'p': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
    'a':
    [
        {
            'i': 'EF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3',
            's': '0',
            'd': 'EF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3'
        }
    ]
}
```

CESR serialization as a Python byte string.

```
(b'-FA_0OKERICAACAAXixnEDmgVuwPOXDjIW3reg4_k8SeJoQEKJKP24fGzeMV4u
KDEDZOA3y_b_0L'
b'G4_cfpKTbWU-_3eeYNM0w9iTkT7frTYsMAABEDZOA3y_b_0LG4_cfpKTbWU-_3e
eYNM0w9iTkT7f'
b'rTYs-JAY-TAXEF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3MAAAEF-
jViYoBr8p3vkp'
b'ZuHlkvxAAY5GZkmQ0QaaHfiE0kg3')
```

Delegated Inception `dip`

Field order by label: `v`, `t`, `d`, `i` , `s`, `kt`, `k`, `nt`, `n`, `bt`, `b`, `c`, `a`, `di`.

Message body as a Python dict.

```
{
    'v': 'KERICAACAACESRAAKM.',
    't': 'dip',
    'd': 'EF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3',
    'i': 'EF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3',
    's': '0',
    'kt': ['1/2', '1/2', '1/2'],
    'k':
    [
        'DEE-HCMSwqMDkEBzlmUNmVBAGIinGu7wZ5_hfY6bSMz3',
        'DHyJFyFzuD5vvUWv5jy6nwWI3wZmSnoePu29tBR-jXkv',
        'DN3JXVEvIjTbisPC4maYQWy6eQIRNdJsxqGFXYUm_ygr'
    ],
    'nt': ['1/2', '1/2', '1/2'],
    'n':
    [
        'EFzr1nnfHpT-nkSfd6vQvbPC-Kq6zy8vbVvUmwxcM1e-',
        'EIXFsLk9kmESy0ZsoHMUaDyK_g3DVRiJQYiAlyeCeYJM',
        'EGVvq4Njkki3EZv838rJrYShBtwXY9o8RUrG2w3nbujn'
    ],
    'bt': '3',
    'b':
    [
        'BFATArhqG_ktVCRLWt2Knbc7JDpaPAFJ4npNEmIW_gPX',
        'BOtF-I9geAUjX9NW1kLIq5qDRNgEXCuwpE4mKHkYuWsF',
        'BEzZUvashpXh_nfPoR6aiqvag0a8E_tbhpeJIgHhOXzl',
        'BCE6biH4a-Zg8LI3cMSx7JRoOvb8rRD62xbyl9N4M2g6'
    ],
    'c': [],
    'a': [],
    'di': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs'
}
```

CESR serialization as a Python byte string.

```
(b'-FCi0OKERICAACAAXdipEF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0k
g3EF-jViYoBr8p'
b'3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3MAAA4AADA1s2c1s2c1s2-JAhDEE-HCM
SwqMDkEBzlmUN'
b'mVBAGIinGu7wZ5_hfY6bSMz3DHyJFyFzuD5vvUWv5jy6nwWI3wZmSnoePu29tBR
-jXkvDN3JXVEv'
b'IjTbisPC4maYQWy6eQIRNdJsxqGFXYUm_ygr4AADA1s2c1s2c1s2-JAhEFzr1nn
fHpT-nkSfd6vQ'
b'vbPC-Kq6zy8vbVvUmwxcM1e-EIXFsLk9kmESy0ZsoHMUaDyK_g3DVRiJQYiAlye
CeYJMEGVvq4Nj'
b'kki3EZv838rJrYShBtwXY9o8RUrG2w3nbujnMAAD-JAsBFATArhqG_ktVCRLWt2
Knbc7JDpaPAFJ'
b'4npNEmIW_gPXBOtF-I9geAUjX9NW1kLIq5qDRNgEXCuwpE4mKHkYuWsFBEzZUva
shpXh_nfPoR6a'
b'iqvag0a8E_tbhpeJIgHhOXzlBCE6biH4a-Zg8LI3cMSx7JRoOvb8rRD62xbyl9N
4M2g6-JAA-JAA'
b'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs')
```

Rotation `rot`

Field order by label: `v`, `t`, `d`, `i`, `s`, `p`, `kt`, `k`, `nt`, `n`, `bt`, `br`, `ba`, `c`, `a`.

Message body as a Python dict.

```
{
    'v': 'KERICAACAACESRAAKs.',
    't': 'rot',
    'd': 'EADBM_Gjzv1_mImlJPPD0bzYmUXmXmCiFIncRYfZMaFc',
    'i': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
    's': '2',
    'p': 'EDmgVuwPOXDjIW3reg4_k8SeJoQEKJKP24fGzeMV4uKD',
    'kt': '2',
    'k':
    [
        'DLv9BlDvjcZWkfPfWcYhNK-xQxz89h82_wA184Vxk8dj',
        'DCx3WypeBym3fCkVizTg18qEThSrVnB63dFq2oX5c3mz',
        'DO0PG_ww4PbF2jUIxQnlb4DluJu5ndNehp0BTGWXErXf'
    ],
    'nt': '2',
    'n':
    [
        'EA8_fj-Ezin_Us_gUcg5JQJkIIBnrcZt3HEIuH-E1lpe',
        'EERS8udHp2FW89nmaHweQWnZz7I8v9FTQdA-LZ_amqGh',
        'EAEzmrPusrj4CDKnSFQvhCEW6T95C7hBeFtZtRD7rOTg'
    ],
    'bt': '4',
    'br': ['BA4PSatfQMw1lYhQoZkSSvOCrE0Sdw1hmmniDL-yDtrB'],
    'ba':
    [
```

```
        'BO3cCAfQiqndZBBxwNk6RGkyA-OA1XbZhBj3s4-VIsCo',
        'BPowpltoeF14nMbU1ng89JSoYf3AmWhZ50KaCaVO6SIW'
    ],
    'c': [],
    'a':
    [
        {
            'i': 'EF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3',
            's': '1',
            'd': 'EFzRkEIXetj-ojZaj0U6P9OqroqZzV0kYwoHGqnlUOwv'
        }
    ]
}
```

CESR serialization as a Python byte string.

```
(b'-FCq0OKERICAACAAXrotEADBM_Gjzv1_mImlJPPD0bzYmUXmXmCiFIncRYfZMa
FcEDZOA3y_b_0L'
b'G4_cfpKTbWU-_3eeYNM0w9iTkT7frTYsMAACEDmgVuwPOXDjIW3reg4_k8SeJoQ
EKJKP24fGzeMV'
b'4uKDMAAC-JAhDLv9BlDvjcZWkfPfWcYhNK-xQxz89h82_wA184Vxk8djDCx3Wyp
eBym3fCkVizTg'
b'18qEThSrVnB63dFq2oX5c3mzDO0PG_ww4PbF2jUIxQnlb4DluJu5ndNehp0BTGW
XErXfMAAC-JAh'
b'EA8_fj-Ezin_Us_gUcg5JQJkIIBnrcZt3HEIuH-E1lpeEERS8udHp2FW89nmaHw
eQWnZz7I8v9FT'
b'QdA-LZ_amqGhEAEzmrPusrj4CDKnSFQvhCEW6T95C7hBeFtZtRD7rOTgMAAE-JA
LBA4PSatfQMw1'
b'lYhQoZkSSvOCrE0Sdw1hmmniDL-yDtrB-JAWBO3cCAfQiqndZBBxwNk6RGkyA-O
A1XbZhBj3s4-V'
b'IsCoBPowpltoeF14nMbU1ng89JSoYf3AmWhZ50KaCaVO6SIW-JAA-JAY-TAXEF-
jViYoBr8p3vkp'
b'ZuHlkvxAAY5GZkmQ0QaaHfiE0kg3MAABEFzRkEIXetj-ojZaj0U6P9OqroqZzV0
kYwoHGqnlUOwv')
```

Delegated Rotation `drt`

Field order by label: `v`, `t`, `d`, `i`, `s`, `p`, `kt`, `k`, `nt`, `n`, `bt`, `br`, `ba`, `c`, `a`, `di`.

Message body as a Python dict.

```
{
    'v': 'KERICAACAACESRAAI4.',
    't': 'drt',
    'd': 'EFzRkEIXetj-ojZaj0U6P9OqroqZzV0kYwoHGqnlUOwv',
    'i': 'EF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3',
    's': '1',
```

```
    'p': 'EF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3',
    'kt': ['1/2', '1/2', '1/2'],
    'k':
    [
        'DB1S8z0h4_qdFhxVHn7BDZb1ErWbBFvcVJX1suKSBctR',
        'DDCDFlbG4dCAX6oIbNffB1mkZqLAS_eHnYUUIPH7BeXB',
        'DP3GAMcSx7eCApzk1N7DceV42o1dZemAe0s3r_-Z0zs1'
    ],
    'nt': ['1/2', '1/2', '1/2'],
    'n':
    [
        'EKUlc5Ml4HLSvdk39k_vh0m6rc061mfM1a4qoEuiBwXW',
        'EJdqHiijmjII-ZtlhFAM5D7myuNeESQkzHoqeWJMMHzW',
        'EDyk8pj0YPHjGNfrG2qZI866WwevwlHEbWYMsKGTGqj2'
    ],
    'bt': '3',
    'br': ['BOtF-I9geAUjX9NW1kLIq5qDRNgEXCuwpE4mKHkYuWsF'],
    'ba': ['BOMrYd5izsqbqaq1WZYa3nbEeTYLPwccfqfhirybKKqx'],
    'c': [],
    'a': []
}
```

CESR serialization as a Python byte string.

```
(b'-FCN0OKERICAACAAXdrtEFzRkEIXetj-ojZaj0U6P9OqroqZzV0kYwoHGqnlUO
wvEF-jViYoBr8p'
b'3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3MAABEF-jViYoBr8p3vkpZuHlkvxAAY5
GZkmQ0QaaHfiE'
b'0kg34AADA1s2c1s2c1s2-JAhDB1S8z0h4_qdFhxVHn7BDZb1ErWbBFvcVJX1suK
SBctRDDCDFlbG'
b'4dCAX6oIbNffB1mkZqLAS_eHnYUUIPH7BeXBDP3GAMcSx7eCApzk1N7DceV42o1
dZemAe0s3r_-Z'
b'0zs14AADA1s2c1s2c1s2-JAhEKUlc5Ml4HLSvdk39k_vh0m6rc061mfM1a4qoEu
iBwXWEJdqHiij'
b'mjII-ZtlhFAM5D7myuNeESQkzHoqeWJMMHzWEDyk8pj0YPHjGNfrG2qZI866Wwe
vwlHEbWYMsKGT'
b'Gqj2MAAD-JALBOtF-I9geAUjX9NW1kLIq5qDRNgEXCuwpE4mKHkYuWsF-JALBOM
rYd5izsqbqaq1'
b'WZYa3nbEeTYLPwccfqfhirybKKqx-JAA-JAA')
```

Receipt Messages

Receipt `rct`

Field order by label: `v`, `t`, `d`, `i`, `s`.

Message body as a Python dict.

```
{
    'v': 'KERICAACAACESRAABw.',
    't': 'rct',
    'd': 'EADBM_Gjzv1_mImlJPPD0bzYmUXmXmCiFIncRYfZMaFc',
    'i': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
    's': '2'
}
```

CESR serialization as a Python byte string.

```
(b'-FAb0OKERICAACAAXrctEADBM_Gjzv1_mImlJPPD0bzYmUXmXmCiFIncRYfZMa
FcEDZOA3y_b_0L'
b'G4_cfpKTbWU-_3eeYNM0w9iTkT7frTYsMAAC')
```

KERI Routed Messages

These have the packet types `qry`, `rpy`, `pro`, `bar`, `exn`

Query Message

Field order by label: `v`, `t`, `d`, `dt`, `r`, `rr`, `q`.

Message body as a Python dict.

```
{
    'v': 'KERICAACAACESRAAD0.',
    't': 'qry',
    'd': 'EF6usM5fNtZWF33E_EQTo9cgU-5f2DH7iBK2V0RPexSe',
    'i': 'EF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3',
    'dt': '2025-08-21T17:50:00.000000+00:00',
    'r': '/oobi',
    'rr': '/oobi/process',
    'q':
    {
        'i': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
        'role': 'witness'
    }
}
```

CESR serialization as a Python byte string.

```
(b'-FA80OKERICAACAAXqryEF6usM5fNtZWF33E_EQTo9cgU-5f2DH7iBK2V0RPex
SeEF-jViYoBr8p'
b'3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg31AAG2025-08-21T17c50c00d000000p
00c006AACAAA-'
b'oobi6AAEAAA-oobi-process-IAQ0J_iEDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM
```

```
0w9iTkT7frTYs'
b'1AAFroleYwitness')
```

Reply Message

Field order by label: `v`, `t`, `d`, `dt`, `r`, `a`.

Message body as a Python dict.

```
{
    'v': 'KERICAACAACESRAAFA.',
    't': 'rpy',
    'd': 'EPvuKFb4DpBKOA-HPJHKXf3mHFokUcYnBE3tjBougM9S',
    'i': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
    'dt': '2020-08-21T17:52:00.000000+00:00',
    'r': '/oobi/process',
    'a':
    {
        'i': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
        'url': 'https://example.com/witness/BGKV6v93ue5L5wsgk75t6
j8TcdgABMN9x-eIyPi96J3B'
    }
}
```

CESR serialization as a Python byte string.

```
(b'-FBP0OKERICAACAAXrpyEPvuKFb4DpBKOA-HPJHKXf3mHFokUcYnBE3tjBougM
9SEDZOA3y_b_0L'
b'G4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs1AAG2020-08-21T17c52c00d000000p
00c006AAEAAA-'
b'oobi-process-IAm0J_iEDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTY
sXurl4BAYaHR0'
b'cHM6Ly9leGFtcGxlLmNvbS93aXRuZXNzL0JHS1Y2djkzdWU1TDV3c2drNzV0Nmo
4VGNkZ0FCTU45'
b'eC1lSXlQaTk2SjNC')
```

Prod Message

Field order by label: `v`, `t`, `d`, `dt`, `r`, `rr`, `q`.

Message body as a Python dict.

```
{
    'v': 'KERICAACAACESRAAEA.',
    't': 'pro',
    'd': 'EJRa0zYQjeupTLGMJxdLBkxZP175elZFCI_Ddg0IjKI1',
    'i': 'EF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3',
    'dt': '2025-08-21T17:50:00.000000+00:00',
```

```
    'r': '/confidential',
    'rr': '/confidential/process',
    'q':
    {
        'i': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
        'name': True
    }
}
```

CESR serialization as a Python byte string.

```
(b'-FA_0OKERICAACAAXproEJRa0zYQjeupTLGMJxdLBkxZP175elZFCI_Ddg0IjK
I1EF-jViYoBr8p'
b'3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg31AAG2025-08-21T17c50c00d000000p
00c006AAEAAA-'
b'confidential6AAGAAA-confidential-process-IAP0J_iEDZOA3y_b_0LG4_
cfpKTbWU-_3ee'
b'YNM0w9iTkT7frTYs1AAFname1AAM')
```

Bare Message

Field order by label: `v`, `t`, `d`, `dt`, `r`, `a`.

Message body as a Python dict.

```
{
    'v': 'KERICAACAACESRAADs.',
    't': 'bar',
    'd': 'EMaAeoTKrRTGIhJeSp-WhwIMSQMvdf13fChMWV6IL6fa',
    'i': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
    'dt': '2020-08-22T17:52:00.000000+00:00',
    'r': '/confidential/process',
    'a':
    {
        'i': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
        'name': 'Ean'
    }
}
```

CESR serialization as a Python byte string.

```
(b'-FA60OKERICAACAAXbarEMaAeoTKrRTGIhJeSp-WhwIMSQMvdf13fChMWV6IL6
faEDZOA3y_b_0L'
b'G4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs1AAG2020-08-22T17c52c00d000000p
00c006AAGAAA-'
b'confidential-process-IAP0J_iEDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9i
```

```
TkT7frTYs1AAF'
b'nameXEan')
```

Exchange Transaction Inception Message

Field order by label: `v`, `t`, `d`, `i`, `ri`, `dt`, `r`, `q`, `a`.

Message body as a Python dict.

```
{
    'v': 'KERICAACAACESRAAE0.',
    't': 'xip',
    'd': 'EISX00jpyZ1_XZBubJghQ2MSxAEgbuBPSoNIKT-4EdwU',
    'u': '0ABrZXJpc3BlY3dvcmtyYXcw',
    'i': 'EF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3',
    'ri': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
    'dt': '2020-08-30T13:30:10.123456+00:00',
    'r': '/offer',
    'q':
    {
        'timing': 'immediate'
    },
    'a':
    {
        'action': 'sell',
        'item': 'Rembrant',
        'price': 300000.0
    }
}
```

CESR serialization as a Python byte string.

```
(b'-FBM0OKERICAACAAXxipEISX00jpyZ1_XZBubJghQ2MSxAEgbuBPSoNIKT-4Ed
wU0ABrZXJpc3Bl'
b'Y3dvcmtyYXcwEF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3EDZOA3y
_b_0LG4_cfpKT'
b'bWU-_3eeYNM0w9iTkT7frTYs1AAG2020-08-30T13c30c10d123456p00c005AA
CAA-offer-IAF'
b'0Mtiming0N_immediate-IAO0Maction1AAFsell1AAFitem1AANRembrant0L_
price4HAC3000'
b'00p0')
```

Exchange Message

Field order by label: `v`, `t`, `d`, `i`, `ri`, `x`, `p`, `dt`, `r`, `q`, `a`.

Message body as a Python dict.

```
{
    'v': 'KERICAACAACESRAAFw.',
    't': 'exn',
    'd': 'ELG8gjElCt6Q53u0m6QuvVRle32EJz0quZkWITml8BMb',
    'i': 'EDZOA3y_b_0LG4_cfpKTbWU-_3eeYNM0w9iTkT7frTYs',
    'ri': 'EF-jViYoBr8p3vkpZuHlkvxAAY5GZkmQ0QaaHfiE0kg3',
    'x': 'EISX00jpyZ1_XZBubJghQ2MSxAEgbuBPSoNIKT-4EdwU',
    'p': 'EISX00jpyZ1_XZBubJghQ2MSxAEgbuBPSoNIKT-4EdwU',
    'dt': '2020-08-30T13:42:11.123456+00:00',
    'r': '/agree',
    'q':
    {
        'timing': 'immediate'
    },
    'a':
    {
        'action': 'buy',
        'item': 'Rembrant',
        'price': 300000.0
    }
}
```


CESR serialization as a Python byte string.


```
(b'-FBb0OKERICAACAAXexnELG8gjElCt6Q53u0m6QuvVRle32EJz0quZkWITml8B
MbEDZOA3y_b_0L'
b'G4_cfpKTbWU-_3eeYNM0w9iTkT7frTYsEF-jViYoBr8p3vkpZuHlkvxAAY5GZkm
Q0QaaHfiE0kg3'
b'EISX00jpyZ1_XZBubJghQ2MSxAEgbuBPSoNIKT-4EdwUEISX00jpyZ1_XZBubJg
hQ2MSxAEgbuBP'
b'SoNIKT-4EdwU1AAG2020-08-30T13c42c11d123456p00c005AACAA-agree-IA
F0Mtiming0N_i'
b'mmediate-IAN0MactionXbuy1AAFitem1AANRembrant0L_price4HAC300000p
0')
```


Out-Of-Band-Introduction (OOBI)

An Out-Of-Band Introduction (OOBI) provides a discovery mechanism that associates a given URI or URL with a given AID or SAID. The URI provided by an OOBI acts as a service endpoint for discovering verifiable information about the AID or SAID. As such, an OOBI itself is not trusted but MUST be verified. To clarify, any information obtained from the service endpoint provided in the OOBI MUST be verified by some other mechanism. An OOBI, however, enables any internet and web search infrastructure to act as an out-of-band infrastructure to discover verifiable information over an in-band mechanism or protocol. The primary in-band verification protocol is KERI. The OOBI protocol provides a web-based bootstrap and/or discovery mechanism for the KERI and the ACDC (Authen-

tic Chained Data Container) protocols ACDC OOBI. Thus, the security (or, more correctly, the lack of security) of an OOBI is out-of-band with respect to a KERI AID or an ACDC that uses KERI. To clarify, everything in KERI or that depends on KERI is end-verifiable; therefore, it has no security dependency, nor does it rely on security guarantees that may or may not be provided by web or internet infrastructure. OOBIs provide a bootstrap that enables what we call Percolated Information Discovery (PID) based on the academic concept called Invasion Percolation Theory [27] [28] [25] [26]. This bootstrap may then be parlayed into a secure mechanism for accepting and updating data. The principal data acceptance and update policy is denoted BADA (Best-Available-Data-Acceptance).

Vacuous discovery of IP resources such as service endpoints associated with a KERI AID or SAID depend an OOBI to associate a given URL with a given AID or SAID SAID OOBI [29]. The principal reason for this dependency is that KERI AIDs are derived in a completely decentralized manner. The root-of-trust of a KERI AID is completely independent of the Internet and DNS addressing infrastructure. Thus, an IP address or URL could be considered a type of Out-Of-Band Infrastructure (OOBI) for KERI for bootstrapping discovery. In this context, an introduction is an association between a KERI AID and a URL that may include either an explicit IP address or a DNS name for its host [RFC3986] [29]. We call this a KERI OOBI (and is a special case of OOBI) with a shared acronym. For the sake of clarity, unless otherwise qualified, OOBI is used to mean this special case of an 'introduction' and not the general case of 'infrastructure'.

Moreover, because IP infrastructure is not trusted by KERI, a KERI OOBI by itself is considered insecure with respect to KERI, and any OOBI must, therefore, be later verified using a KERI BADA mechanism. The principal use case for an OOBI is to jump-start or bootstrap the discovery of a service endpoint for a given AID. To reiterate, the OOBI by itself is not sufficient for discovery because the OOBI itself is insecure. The OOBI merely jump-starts or bootstraps the authenticated discovery.

OOBIs enable a KERI implementation to leverage existing IP and DNS infrastructure to introduce KERI AIDs and discover service endpoints, which may then be securely attributed. KERI does not, therefore, need its own dedicated discovery network; OOBIs with URLs will do.

A secondary use case for OOBIs is to provide service endpoints or URIs for SAD (items identifier by their SAID). A SAID is a content address derived from a cryptographic digest of the serialization of a data item. The SAID protocol [1] provides a derivation process where the SAID is actually included in the SAD. This makes a SAID self-referential. Verification of a SAD resource obtained by querying a URI that includes the SAD's SAID is accomplished by simply re-deriving the SAID of the SAD in the reply and comparing it to the SAID in the URI. The `sad` URI scheme may be simply expressed as `sad:said`

where `said` is replaced with the actual SAID of the referenced SAD item. The media type of the returned SAD is determined by its CESR-compatible serialization type, such as JSON, CBOR, MGPK, or native CESR, for example.

Basic OOBI

The simplest form of a KERI OOBI MAY be expressed by any of a namespaced string, a tuple, a mapping, a structured message, or a structured attachment where every form contains both a KERI AID and a URL (or URI). The OOBI associates the URL with the AID. By convention, the URL typically includes the word `oobi` in its path to indicate that it is to be used as an OOBI, but this is NOT REQUIRED. In abstract tuple form, an OOBI is as follows:

```
(url, aid)
```

In concrete tuple form, an OOBI is as follows:

```
("http://8.8.5.6:8080/oobi", "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjB
UcMVtvhmB")
```

An OOBI itself is not signed or otherwise authenticatable by KERI but may employ some other Out-Of-Band-Authentication (OOBA) mechanism, i.e., non-KERI.

The OOBI is intentionally simplistic to enable very low byte count introductions such as a may be conveyed by a QR code or Data matrix [29] [30].

OOBI URL (IURL)

URLs provide a namespace, which means that the mapping between URL and AID can be combined into one namespaced URL where the AID is in the path component and any other hints, such as roles or names, are in the query component of the URL. This would be a type of self-describing OOBI URL.

For example, suppose the AID is

```
EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB
```

This may be included as a path component of the URL, such as,

```
http://8.8.5.6:8080/oobi/EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVt
vhmB
```

This is called an OOBI URL, or IURL for short. All that is needed to bootstrap the discovery of a KERI AID is an IURL. KERI can leverage the full IP/DNS infrastructure as a discovery bootstrap of an AID by providing an associated IURL.

The AID may act in any of the KERI roles such as `watcher`, `witness`, `juror`, `judge` or `registrar` but is usually a `controller`. In the latter case, the IURL may be a service endpoint provided by one of the supporting components for a given controller. Thus, the AID in an OOBI may be either a controller ID, CID or an endpoint provider ID, EID. The resource at that URL in the OOBI is ultimately responsible for providing that detail, but an OOBI as a URL may contain hints in the query string for the URL, such as a `role` and/or `name` designation.

```
https://example.com/oobi/EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVt
vhmB?role=witness
```

```
http://8.8.5.6:8080/oobi/EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVt
vhmB?role=watcher&name=eve
```

When the role is provided in the IURL, the EID of the endpoint provider for that role would be discovered via the proof returned by querying the URL. In addition, the proof returned may indicate a different URL for that role, so a self-describing IURL may also act as a forwarding mechanism.

To clarify, the minimum information in an OOBI is the pair `(URL, AID)`. The compact representation of an OOBI leverages the namespacing of the URL itself to provide the AID. Furthermore, the query string in the URL namespace may contain other information or hints, such as the role of the service endpoint represented by the URL and/or a user-friendly name.

Well-Known OOBI

An OOBI may be returned as the result of a 'GET' request to an [spec: RFC5785] well-known URL.

For example,

```
 /.well-known/keri/oobi/EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtv
hmB
```

Where: `EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB` is the AID and the result of the request is either the target URL or a redirection to the target URL, where the target URL can be found, such as,

```
https://example.com/witness/witmer
```

or

```
http://8.8.5.5:8080/witness/witmer
```

The resultant target URL may be in a different domain or IP address from the `well-known` resource.

CID and EID

A more verbose version of an OOBI would also include the endpoint role and the AID (EID) of the endpoint provider in a self-describing OOBI URL. And endpoint provider might by a Witness .

For example,

```
https://example.com/oobi/EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVt
vhmB/witness/BGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3B
```

where `EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB` is the AID (CID) of the controller and `BGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3B` is the AID (EID) of one of its witnesses as endpoint provider.

Similarly,

```
http://8.8.5.6/oobi/EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB/
witness/BGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3B
```

Where: where `EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB` is the AID (CID) of the controller and `BGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3B` is the AID (EID) of one of its witnesses as endpoint provider.

Multi-OOBI (MOOBI)

An OOBI may include a list of URLs, thus simultaneously making an introductory association between the AID and multiple URLs. This would be a multi-OOBI (MOOBI). In general, an MOOBI is a special case of an OOBI without making a named distinction. The first KERI reply message below is an example of a MOOBI.

KERI Reply Messages as OOBIs

A more verbose expression for an OOBI would be an unsigned KERI reply message, `rpy`. The route, `r` field in the message starts with `/oobi`. This specifies that it is an OOBI, so the recipient knows to apply OOBI processing logic to the message. A list of URLs may be provided so that one reply message may provide multiple introductions. In the following examples, the reply messages are serialized with JSON.

Reply message as Python dict.

```
{
    "v": "KERICAACAAJSONAAIA.",
    "t": "rpy",
    "d": "ELtIQ71PMr9m5a8eYiC39hikuU8yTWoFw1vWjtqbVUX4",
    "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "dt": "2020-08-21T17:52:00.000000+00:00",
    "r": "/oobi/witness",
    "a":
    {
        "cid": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
        "urls":
        [
            "https://example.com/witness/wilma/BGKV6v93ue5L5wsgk7
5t6j8TcdgABMN9x-eIyPi96J3B",
            "https://example.com/witness/watson/BAPv2MnoiCsgOnklm
FyfU07QDK_93NeH9iKfOy8V22aH",
            "https://example.com/witness/winona/BA4PSatfQMw1lYhQo
ZkSSvOCrE0Sdw1hmmniDL-yDtrB"
        ]
    }
}
```

Serialized reply message as a Python byte string of JSON without whitespace.

```
(b'{"v":"KERICAACAAJSONAAIA.","t":"rpy","d":"ELtIQ71PMr9m5a8eYiC3
9hikuU8yTWoFw1'
b'vWjtqbVUX4","i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhm
B","dt":"2020-08'
b'-21T17:52:00.000000+00:00","r":"/oobi/witness","a":{"cid":"EPR7
FWsN3tOM8PqfM'
b'ap2FRfF4MFQ4v3ZXjBUcMVtvhmB","urls":["https://example.com/witne
ss/wilma/BGKV'
b'6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3B","https://example.com/
witness/watso'
b'n/BAPv2MnoiCsgOnklmFyfU07QDK_93NeH9iKfOy8V22aH","https://exampl
e.com/witness'
b'/winona/BA4PSatfQMw1lYhQoZkSSvOCrE0Sdw1hmmniDL-yDtrB"]}}')
```

A service endpoint location reply message could also be re-purposed as an OOBI by using a special route path that starts with `/oobi` but also includes the AID being introduced and, optionally, the role of the service endpoint provider. This approach effectively combines the information from both the `/end/role` and `/loc/scheme` reply messages into one. This may allow a shortcut to authenticate the service endpoint. This is shown below.

Reply message as Python dict.

```
{
    "v": "KERICAACAAJSONAAFq.",
    "t": "rpy",
    "d": "EFMQh0w5-AHw-H01DtqEFhAIC6KXbjYvUSOEX6kSPY4j",
    "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "dt": "2020-08-21T17:52:00.000000+00:00",
    "r": "/oobi/EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB/witn
ess",
    "a":
    {
        "eid": "BGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3B",
        "scheme": "https",
        "url": "https://example.com/witness/wilma"
    }
}
```


Serialized reply message as a Python byte string of JSON without whitespace.


```
(b'{"v":"KERICAACAAJSONAAFq.","t":"rpy","d":"EFMQh0w5-AHw-H01DtqE
FhAIC6KXbjYvUS'
b'OEX6kSPY4j","i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhm
B","dt":"2020-08'
b'-21T17:52:00.000000+00:00","r":"/oobi/EPR7FWsN3tOM8PqfMap2FRfF4
MFQ4v3ZXjBUcM'
b'VtvhmB/witness","a":{"eid":"BGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-e
IyPi96J3B","s'
b'cheme":"https","url":"https://example.com/witness/wilma"}}')
```


Self (Blind) OOBI (SOOBI)

A bare URL but no AID may be used as a self (blind) OOBI for blind or self-introductions e.g., SOOBI. Querying that SOOBI may return or result in a default target OOBI or default target endpoint reply. This provides a mechanism for self-introduction or blind i.e., self OOBI (SOOBI). Consider the examples of self-OOBIs below.

```
http://8.8.5.7:8080/oobi

http://localhost:8080/oobi

http://8.8.5.7:8080/oobi?role=controller&name=eve

http://localhost:8080/oobi?role=controller&name=eve
```

To elaborate, by default, the result of a `GET` request to a self OOBI URL could be another OOBI with an AID that is the `self` AID of the node providing the self OOBI endpoint or the actual authenticatable `self` endpoint with its AID or a default set of authenticat-

able endpoints. This is useful to bootstrap components in an infrastructure where the target URLs do not use a public DNS address but instead use something more secure like an explicit public IP address or a private IP or private DNS address. A self-introduction provides a bootstrap mechanism similar to a hostname configuration file with the exception that in the OOBI case, the AID is not in the configuration file, just the bare URL, and the given node queries that bare URL (SOOBI) to get the target endpoint AID. This allows bootstrap using bare IP addresses in systems where the IP infrastructure is more securely managed than public DNS or where some other OOBA mechanism is used in concert.

To clarify, because a bare URL, acting as a SOOBI, does not expose an AID, the resultant response when querying the OOBI may depend on other factors such as the source IP of the querier (requester) and/or another OOBA mechanism. This supports the private bootstrap of infrastructure. Of course, one could argue that this is just kicking the can down the road, but IP addresses are correlatable, and a self OOBI can leverage IP infrastructure for discovery when used in combination with some other OOBA mechanism without unnecessary correlation.

For example, a given indirect mode controller is identified by its AID (CID). The controller MUST also create witness hosts with endpoints. This means first spinning up witness host nodes and creating witness AIDs (WIDs) for those nodes. Given that these WIDs MUST be eventually designated in the KEL for the CID, the controller of the CID can confirm using its KEL that the signed endpoint reply provided by a SOOBI request is indeed signed by the corresponding private keys for a WID designated in its KEL. This means that the only place that the WID MUST appear is in the KEL and not in all the config files used to bootstrap communications between the CID host and its designated WID hosts. Just SOOBIs will do. Whereas regular OOBIs with redundant configuration information may be a vector for a type of DDOS attack where corrupted, inconsistent, redundant configuration information results in a failure to boot a system. Redundancy for security is best applied in the context of a self-healing or resilient threshold structure that explicitly manages the redundancy as a security mechanism instead of as un-managed inadvertent redundancy.

OOBI Forwarding

In every case, an OOBI may result in proof for a different URL than that provided in the OOBI itself. It allows OOBI forwarding so that introductions produced as hard copies, such as QR codes, do not necessarily become stale. The recipient of the OOBI may choose to accept that proof or not. Ultimately, the recipient only treats URLs as valid endpoints when they are fully KERI authenticated. Given that an OOBI result is always KERI authenticated before use in a given role, the worst case from a security perspective is that an OOBI may be part of a DDOS attack but not as part of a service endpoint cache poison attack.

OOBI with MFA

An OOBI may be augmented with one or more OOBAs to minimize the likelihood of a DDOS OOBI attack. A given recipient may require as a precondition to accepting an OOBI one or more OOBA mechanisms, such as text messages, emails, etc., that provide some degree of non-KERI-based security to the OOBI. Thus, an OOBI could employ out-of-band (with respect to KERI) multi-factor authentication (MFA) to preclude any OOBI-based DDOS attacks on KERI.

SPED (Speedy Percolated Endpoint Discovery)

All the information needed to discover and verify is bootstrapped from the OOBI. Subsequent authorization is non-interactive, thus making it highly scalable. BADA-RUN authorization is also lightweight for the host because the only memory requirements are a sequence number, date-time stamp window, and nullification state. This provides what we call zero-trust percolated discovery or speedy percolated endpoint discovery (SPED) [25][26][27][28]. Percolation means that each discoverer, in turn, may share what it discovers with any subsequent discoverers. Because the information so discovered is end-verifiable, the percolation mechanism does not need to be trusted. Percolating intermediaries do not need to be trusted.

JIT/NTK Discovery

With percolated discovery, discovery mechanisms can be made very efficient because they can be optimized for any given exchange of verifiable data that requires discovery. This is called just-in-time/need-to-know JIT/NTK discovery. Each Exchanger of verifiable data MUST have already verified the data before exchanging it with the Exchangee. Therefore, all the information needed to verify (proofs) MUST have already been available to the Exchanger, i.e., need-to-know. The Exchanger can then percolate that verification information to the Exchangee at the time of exchange, i.e, just-in-time. This avoids the need to have a dedicated global infrastructure for the discovery of verifiable data and the associated proofs.

Summary

The main value of an OOBI is that it is compact and is not encumbered by authentication proofs but may be used to kick-start the process of authentication (proving).

One way to pre-configure a vacuous KERI installation is to provide OOBIs or SOOBIs in a configuration file. The bootstrap process of the installation then queries the associated URLs to retrieve the KERI authentication proofs (BADA) that are then used to populate its database securely. This simplifies the configuration file.

An alternative would be to populate the configuration file with the KERI authentication proofs. But these proofs may be quite verbose and cumbersome and may make the config file somewhat difficult to manage in human-readable/writable form. Furthermore, if one already had the proofs, one could just pre-populate the database with those proofs.

Therefore OOBI, OOBI-based configuration files may be advantageous as either easier to manage or as a viable option when the proofs are not yet available at configuration time.

Furthermore, a clean clone replay restart of a given KERI component is designed to fix any unverified corruption of its associated KELs. If each component uses OOBIs to retrieve the authentication proofs from other components, then all the components will have clean proofs instead of stale proofs.

BADA (Best-Available-Data-Acceptance) Policy

The recipient of an OOBI verifies the OOBI by authenticating the endpoint URL given by the OOBI with respect to an authorization signed by the controller of the AID given by the OOBI. This authorization follows the BADA policy. The BADA policy guarantees the monotonicity of updates to authentically signed data at rest. This follows best practices for zero-trust computing infrastructure for authentic data. The authorization is usually obtained as a resource in reply to a query to the OOBI URL. Specifically, the service endpoint at the URL responds with a resource that contains the supporting reply messages that are KERI authenticatable.

Security Issues

KERI follows a “zero-trust” security model for authentic or securely attributable data. That means that data is signed both in motion and at rest. The primary attack against signed data is a replay attack. In a replay attack, an adversary obtains a copy of data with a verifiable signature and then replays it later. Without some other information, it is difficult for a host to detect that it is indeed a replay or malicious reuse of signed data and not the original use of that data.

To elaborate, there are two primary types of attacks on authentic or authenticatable data-at-rest. The first is a replay attack. The second is a deletion attack. In a replay attack, an adversary keeps a copy of an authentic message or data together with its verifiable signature that has already been created and used by the controller of a KERI AID and then sometime later replays that same message with the signature. A verifier may thereby be fooled into believing that the replay is actually a new message and not a stale message. There are both interactive and non-interactive mitigations to replay attacks. Interactive mitigations use some type of nonce or Salt exchanged between Updater and Updatee. The nonce exchange introduces latency, scalability, and synchronization limitations. Non-interactive mitigations require a monotonic ordering mechanism. Typically, monotonic ordering is based on logic rooted in a sequence number or date-time stamp. Because non-interactive mitigations are asynchronous, however, they do not have the latency and scalability limitations of interactive mitigations and are therefore preferred.

The KEL of a KERI AID provides such a monotonic ordering mechanism as it employs both a sequence number and digest chaining. For authentic data directly anchored to or determined by a KEL, the relative KEL location determines the monotonic order. This ordering determination includes TELs , which themselves are monotonically ordered with respect to anchoring seals in the associated KEL ACDC . For authentic data not directly anchored or included in a KEL, the relative key state (which is determined by the KEL) may be used in combination with a date-time stamp to ensure monotonic ordering. Finally, for any AID whose key state is fixed, a date-time stamp may be used with appropriate update logic to ensure monotonic ordering. The logic that ensures monotonic ordering is called BADA and is described later in this section.

A deletion attack is related to a replay attack. Once erased or deleted, a verifier may not be able to detect a replay attack of the deleted data because it has lost a record of the prior play to compare against. To elaborate, once erased, any stale authenticated data acting as authorization may be replayed without detection. This exposes a problem with the GPDR (General Data Protection Regulation) right-to-erasure, which, if naively implemented as total erasure, exposes the data controller to a replay attack of erased data.

The primary mitigation mechanism for deletion attacks is maintaining redundant copies of the signed authentic data. If one of the redundant copies has not been deleted, then comparing the hosts of the redundant copies will expose the deletion attack. The monotonicity of the data is preserved in each copy. The hosts need merely compare copies. Only the current data item needs to be kept in full in order to support the use of that data. For protection against replay attacks using stale data, only copies of the digest or signature of the data MUST be kept. To reiterate, a replay attack can be detected by comparing the digest or signature (which is a type of digest) of any undeleted copy with the presented data.

To summarize, authentic data at rest consists of the data item and signature(s). The two primary attacks are replay and deletion. Replay attack mitigation relies on replay monotonicity in data updates. Deletion attack mitigation relies on the redundancy of monotonic data.

BADA Rules

The BADA rules apply to any data item stored in a database record whose value is used for some defined purpose. Updates are sourced from the controller of an associated KERI AID. The primary purpose of BADA policy is to enforce monotonicity of the updates with respect to the key state of that associated AID. This primarily protects against replay attacks on the database record. For example, a rollback to an earlier value via replay of an earlier update. An Update or change to the database record is ‘accepted’ when it follows the BADA rules (policy) for acceptance. The BADA rules ensure the monotonicity of all updates.

There are two different mechanisms for the controller of an AID to authorize updates to a given database record. The first is by referencing the update in the KEL of the authorizing AID. All entries in a KEL MUST be signed by the current signing keypair(s) given by the Key-state for that KEL. The second is by signing a date-time stamped update. In this case, the update MUST either includes a reference to the Key-state in the authorizing AID's KEL from which the signing keypair(s) needed to verify the signature is obtained or the AID MUST be ephemeral with a fixed Key-state (has a non-transferable derivation code). The rules differ for each of the two mechanisms.

KEL Anchored Updates

In this case, the update to a record is included in or anchored via a seal to the AID's Key-state in its KEL. In either case, the update is referenced in an event in the KEL of the AID. By virtue of the reference, the Controller of that KEL's AID is authorizing that Update. The record MAY have a Prior value that is being updated or the Update MAY serve to create the initial value of the record. Prior means the prior record.

```
Rules for the acceptance of the Update:  (in order of priority)
  Confirm Update is anchored or included in AID's KEL.

  WHEN Update is anchored in AID's KEL AND...
    IF no Prior THEN accept. (always)
    IF Prior AND...
  The Update's anchor appears later in KEL than the Prior's ancho
r THEN accept.
  Otherwise, do not accept.
```

Signed (Not Anchored) Updates

In this case, the update to a record is signed by the controller of the AID, but the update itself is NOT included in or anchored to the AID's KEL. The record may have a Prior value that is being updated or the update serves to create the initial value of the record. In this context, Prior means the Prior record. All date times are relative to the controller's date time, NOT the database host's date time.

There are two cases. These are as follows.

1. Ephemeral AID whose Key-state is fixed (no KEL needed)
2. Persistent AID whose Key-state is provided by a KEL

```
Rules for the acceptance of the Update: (in order of priority)

Confirm signature on the Update verifies against indicated key-st
ate under which signature was made.

  WHEN signature verifies AND...
    IF no Prior THEN accept (always).
```

```
    IF Prior THEN ...
    Compare the Update’s verified signature key-state against the
Prior's verified signature key-state.
      IF the Update’s key-state appears later in KEL than the Pri
or's key-state THEN accept.
      IF both the Update’s and the Prior's key-states appear at t
he same location in KEL AND...
        the Update’s date-time is later than the Prior's date-tim
e THEN accept.
  Otherwise, do not accept.
```

RUN off the CRUD

In the conventional client-server database architecture, the database server is responsible for creating records on behalf of clients and assigning unique identifiers for each record. The server returns to the client the unique record identifier when it creates a record. The server is the source of truth. But in a zero-trust (end-verifiable) decentralized peer-to-peer architecture, there is no client/server. Every host is a Peer. Each Peer MUST be the source of truth for its own data. Therefore, each Peer is responsible for managing its own records. Each Peer MUST be able to create unique identifiers for its own data. This inverts the architecture because each Peer creates a unique identifier for each of its own data items and sends that identifier with the data item to the other Peers. Each Peer stores data on behalf of the other Peers. This inverted architecture enables consistent, authentic data update policies that work asynchronously across multiple Peers and are replay and deletion attack-resistant. Each Peer has an end-verifiable (via signature) monotonically updated view of the data records sourced from the other Peers.

The acronym for the traditional client-server database update policy is CRUD (Create, Read, Update, Delete). The acronym for this new peer-to-peer end-verifiable monotonic update policy is RUN (Read, Update, Nullify). As described above, because the source of truth for each data item is a decentralized controller Peer, a given database hosted by any Peer does not Create records in the traditional sense of a server creating records for a client. The hosting Peer merely stores a copy of an Update to records sent out by the source Peer (controller). Thus, there is no Create action, only an Update action. When a Peer has not yet seen any version of a record, then its copy is vacuous and is replaced by the first Update it sees. To clarify, a source Peer Updates other Peers by sending out the latest copy or version of its own record. The original copy or version is always created by the source Peer not the data hosting destination Peer or destination Peer for short.

In order to ensure that the destination Peers are resistant to replay and deletion attacks, destination Peers apply non-interactive monotonic update logic to any Updates they receive from the source Peer. This means that a destination Peer MUST NOT ever delete a record storing the latest version of an Update. Thus, there is no Delete. Instead of Delete, Peers Nullify. A Nullify is a special type of Update that indicates that the data in the record is no longer valid but, importantly, without actually erasing or Deleting the

record that includes a reference to the latest monotonic determining anchor and/or date-time. There are two ways to indicate Nullification. The first is to assign a `null` value to the record. This works for single-field records. The second is to attach an associated Boolean logic flag field that indicates the record has been Nullified. This works for multi-field records.

OOBI KERI Endpoint Authorization (OKEA)

An important use case for BADA-RUN is to process OOBIs that provide service endpoint discovery of the AIDS of KERI components. These components include but are not limited to Controllers, Agents, Backers (Witness or Registrar), Watchers, Jurors, Judges, and Forwarders. An endpoint is a URL that may include an IP Scheme, Host, Port, and Path. The model for securely managing endpoint data starts with a Principal Controller of an AID. A Principal Controller authorizes some other component to act as a Player in a Role. Typically, a Role serves some function needed by the Principal Controller to support its AID and may be reached at a service endpoint URL for that Role. Each component, in turn, is the Controller of its own AID. Each component AID is a Player that may provide or act in a Role on behalf of the Principal Controller by providing services at the associated service endpoint for its associated Role.

The authorization model uses a zero-trust BADA-RUN policy to Update authorizations. A Principal Controller authorizes a Player by signing a Role authorization message that authorizes the Player's AID to act in a role. A Player authorizes its endpoint URL by signing an endpoint authorization message that authorizes a URL (location) with a scheme. Any Peer may keep an updated copy of the latest service endpoint URL(s) provided by a Player in a Role for a given Principal AID by following the BADA-RUN policy on Updates sent to its database of these authorizations. The authorizations are issued in the context of the KERI Key-state for the Principal and Player AIDs.

Some components (Players in Roles) are implicitly authorized by the Principal Controller by being explicitly designated in the KEL of the Principal, i.e., there is no explicit authorization message of the Player/Role. The authorization is implied by the KEL entry. For example, a Backer designation of a Witness or Registrar AID implicitly authorizes that AID to act as a Player in the Role of Witness or Registrar. An associated explicit endpoint authorization message signed by that Witness or Backer is still needed to provide the URL (location and scheme) of the actual service endpoint for that Player.

The combination of KERI and the BADA-RUN policy enables any Controller to manage data in a zero-trust architecture at the database level. Any Controller may promulgate verifiably authentic information with replay and deletion attack resistance. The authentic information may be simply source data or instead authorizations to enable some other function. The hard work of determining the associated Key-state is provided by KERI. KERI makes the establishment of authenticity straightforward. The BADA-RUN policy protects against replay and deletion attacks given authentic data.

This approach follows the many thin layers approach of the Hourglass protocol model. BADA-RUN is a thin layer on top of KERI authenticity. OOBIs are a thin discovery layer that sits on top of a thin authorization layer (leveraging reply messages and BADA-RUN logic) on top of KERI.

This also follows the design ethos of KERI of minimally sufficient means. OOBIs leverage the existing Internet discovery mechanisms but without needing to trust the Internet security model (or the lack of one). End-verifiability in KERI provides safety to any OOBI discovery. The Internet's discovery mechanism, DNS/CA, is out-of-band with respect to KERI security guarantees. Thus, OOBIs may safely use DNS/CA, web search engines, social media, email, and messaging as discovery mechanisms. The worst case is the OOBI fails to result in a successful discovery and some other OOBI is needed.

Typically, the query of a ReST endpoint given by the OOBI URL could return as proof any associated authorizing reply message(s) and any associated KELs.

Authorized Endpoint Disclosure Example

This section provides an example of using OKEA (OOBI KERI Endpoint Authorization) with BADA-RUN for endpoint disclosure.

The KERI protocol defines a generic `reply` message for updating information using the BADA-RUN policy. Each `reply` message includes a route, `r`, field that indicates both the payload type and the handler that SHOULD process the message. The route, `r`, field value is a slash, `/` delimited pathname string. The Principal Controller AID is indicated by the CID (Controller ID) or `cid` field. The endpoint component Player is indicated the EID (Endpoint Controller ID) or `eid` field. There are two different authorization cases. In one case, a CID authorizes an EID in a Role. In the other case, an EID authorizes a Loc (URL location) for a scheme. There are two routes for each type of authorization. One route Updates the authorization and the other Nullifies the authorization. These are summarized as follows,

- Datetime stamped BADA authorization Reply message by CID of EID in Role (Update)
- Datetime stamped BADA deauthorization by CID of EID in Role (Nullify)
- Datetime stamped BADA authorization by EID of URL for scheme (Update).
- Datetime stamped BADA deauthorization by EID of URL for scheme (Nullify)

A party interested in discovering the service endpoint for a given Controller AID initiates the discovery by providing an OOBI. A successful discovery will result in the return of signed `reply` messages that provide verifiable proof that the service endpoint (either directly provided in the OOBI, or indirectly provided via forwarding) is an authorized endpoint for that AID.

To summarize, upon acceptance of an OOBI the recipient queries the provided URL for proof that the URL is an authorized endpoint for the given AID. The proof format may depend on the actual role of the endpoint. A current witness for an AID is designated in the current key state's latest Establishment event in the AID's KEL. Therefore, merely replying with the Key State or KEL may serve as proof for a witness introduced by an OOBI. The actual URL may be authorized by an attendant signed `/loc/scheme` reply message with the URL.

Other roles that are not explicitly part of Key-state (i.e., are not designated in KEL establishment events) MUST be authorized by explicitly signed reply messages. Typically, these will be a signed `/end/role/` reply message. The actual URL may be authorized by an attendant signed `/loc/scheme` reply message with the URL.

Example reply messages.

Player EID in Role by CID Update

Reply message as Python dict.

```
{
    "v": "KERICAACAAJSONAAFI.",
    "t": "rpy",
    "d": "EBcL5FQ2cHPcLmGb7AKk-0Rtq0_A-m-mQTygGxTrqTBb",
    "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "dt": "2020-08-21T17:52:00.000000+00:00",
    "r": "/end/role/add",
    "a":
    {
        "cid": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
        "role": "witness",
        "eid": "BGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3B"
    }
}
```

Serialized reply message as a Python byte string of JSON without whitespace.

```
(b'{"v":"KERICAACAAJSONAAFI.","t":"rpy","d":"EBcL5FQ2cHPcLmGb7AKk
-0Rtq0_A-m-mQT'
b'ygGxTrqTBb","i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhm
B","dt":"2020-08'
b'-21T17:52:00.000000+00:00","r":"/end/role/add","a":{"cid":"EPR7
FWsN3tOM8PqfM'
b'ap2FRfF4MFQ4v3ZXjBUcMVtvhmB","role":"witness","eid":"BGKV6v93ue
5L5wsgk75t6j8'
b'TcdgABMN9x-eIyPi96J3B"}}')
```

Player EID in Role by CID Nullify Via Cut

To nullify the EID use cut route.

Reply message as Python dict.

```
 {
    "v": "KERICAACAAJSONAAFI.",
    "t": "rpy",
    "d": "EH4uEDQHtCxoJ-RXbvmIjl-NE3JoPJ26fN7sZm9dsqPv",
    "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "dt": "2020-08-21T17:52:10.000000+00:00",
    "r": "/end/role/cut",
    "a":
    {
        "cid": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
        "role": "witness",
        "eid": "BGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3B"
    }
}
```

Serialized reply message as a Python byte string of JSON without whitespace.

```
(b'{"v":"KERICAACAAJSONAAFI.","t":"rpy","d":"EH4uEDQHtCxoJ-RXbvmI
jl-NE3JoPJ26fN'
b'7sZm9dsqPv","i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhm
B","dt":"2020-08'
b'-21T17:52:10.000000+00:00","r":"/end/role/cut","a":{"cid":"EPR7
FWsN3tOM8PqfM'
b'ap2FRfF4MFQ4v3ZXjBUcMVtvhmB","role":"witness","eid":"BGKV6v93ue
5L5wsgk75t6j8'
b'TcdgABMN9x-eIyPi96J3B"}}')
```

Endpoint Location with Scheme by EID Update

Reply message as Python dict.

Serialized reply message as a Python byte string of JSON without whitespace.

```
{
    "v": "KERICAACAAJSONAAE6.",
    "t": "rpy",
    "d": "ELH2kZK9QXgV9utSqRE-jf2Xwk4rgca6xk35Mpo4EeZP",
    "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "dt": "2020-08-21T17:52:11.000000+00:00",
    "r": "/loc/scheme",
    "a":
    {
        "eid": "BGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3B",
```

```
            "scheme": "https",
            "url": "https//example.com/witness/wilma"
        }
}
```

Serialized reply message as a Python byte string of JSON without whitespace.


```
(b'{"v":"KERICAACAAJSONAAE6.","t":"rpy","d":"ELH2kZK9QXgV9utSqRE-jf2Xwk4rgca6xk'
b'35Mpo4EeZP","i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB","dt":"2020-08'
b'-21T17:52:11.000000+00:00","r":"/loc/scheme","a":{"eid":"BGKV6v93ue5L5wsgk75'
b't6j8TcdgABMN9x-eIyPi96J3B","scheme":"https","url":"https//example.com/witnes'
b's/wilma"}}')
```


Endpoint Location with Scheme by EID Nullify Via Empty URL

To Nullify set the `url` to the empty string `""`.

Reply message as Python dict.

```
{
    "v": "KERICAACAAJSONAAEa.",
    "t": "rpy",
    "d": "EGWrf4ve6Nlec3iC7ba0-f6YBIHXKRzrGG-bWE-gcHY_",
    "i": "EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB",
    "dt": "2020-08-21T17:52:12.000000+00:00",
    "r": "/loc/scheme",
    "a":
    {
        "eid": "BGKV6v93ue5L5wsgk75t6j8TcdgABMN9x-eIyPi96J3B",
        "scheme": "https",
        "url": ""
    }
}
```

Serialized reply message as a Python byte string of JSON without whitespace.


```
(b'{"v":"KERICAACAAJSONAAEa.","t":"rpy","d":"EGWrf4ve6Nlec3iC7ba0-f6YBIHXKRzrGG'
b'-bWE-gcHY_","i":"EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB","dt":"2020-08'
b'-21T17:52:12.000000+00:00","r":"/loc/scheme","a":{"eid":"BGKV6v93ue5L5wsgk75'
b't6j8TcdgABMN9x-eIyPi96J3B","scheme":"https","url":""}}')
```

Bibliography

Normative section

1. Samuel M. Smith, Composable Event Streaming Representation (CESR), 2022

2. C. Bormann, P. Hoffman, Concise Binary Object Representation (CBOR), 2020

3. Sadayuki Furuhashi, MessagePack, 2008

4. Samuel M. Smith, Key Event Receipt Infrastructure, 2021

30. RFC0791 Internet Protocol. J. Postel; 1981-09. Status: Internet Standard.

31. RFC3986 Uniform Resource Identifier (URI): Generic Syntax. T. Berners-Lee; R. Fielding; L. Masinter; 2005-01. Status: Internet Standard.

32. RFC4627 The application/json Media Type for JavaScript Object Notation (JSON). D. Crockford; 2006-07. Status: Informational.

33. RFC5280 Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. D. Cooper; S. Santesson; S. Farrell; S. Boeyen; R. Housley; W. Polk; 2008-05. Status: Proposed Standard.

34. RFC5785 Defining Well-Known Uniform Resource Identifiers (URIs). M. Nottingham; E. Hammer-Lahav; 2010-04. Status: Proposed Standard.

35. RFC6960 X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. S. Santesson; M. Myers; R. Ankney; A. Malpani; S. Galperin; C. Adams; 2013-06. Status: Proposed Standard.

36. RFC8948 Concise Binary Object Representation (CBOR). C. Bormann; P. Hoffman; 2020-12. Status:Internet Standard

37. RFC4648 The Base16, Base32, and Base64 Data Encodings. S. Josefsson; 2006-10. Status: Proposed Standard.

38. IETF RFC-2119 Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. 1997-03. Status: Best Current Practice

39. ISO/IEC 7498-1:1994 Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model. June 1999. Introduction. Retrieved 26 August 2022.

40. IETF RFC-3339 Date and Time on the Internet: Timestamps DateTime. G. Klyne. 2002-07. Status: Standards Track

41. Blake3 Specification Blake3. J. O'Connor; J-P. Aumasson; S. Neves ; Z. Wilcox-O'Hearn. Version 20211102173700.

Informative section

5. Samuel M. Smith, Universal Identifier Theory, 2020

6. Samuel M. Smith, Decentralized Autonomic Data (DAD) and the three R's of Key Management, 2018

7. David Wilkinson, Jorge F Willemsen, Invasion percolation: a new form of percolation theory, 1983

8. Information-Theoretic and Perfect Security

9. Cryptographically-secure pseudorandom number generator

10. Information Theory

11. Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete

12. Jean-Philippe Aumasson, Too Much Crypto, 2021

13. One-way Function

14. One-way Function

15. Public-key Cryptography

16. Marc Girault, Self-certified public keys

17. M. Kaminsky, E. Banks, SFS-HTTP: Securing the Web with Self-Certifying URLs, 1999

18. David Mazieres, Self-certifying File System, 2000

19. David Mazieres, M. Kaashoek, Escaping the Evils of Centralized Control with self-certifying pathnames, 2000

20. Certificate Revocation List

21. Verifiable Data Structures

22. Ricardian contract

23. Namespace

24. Eclipse Attack

25. Percolation Theory

26. First Passage Percolation

27. Invasion Percolation

28. Uniform Resource Locator

29. QR Code

30. Data Matrix