Composable Event Streaming Representation

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

---

CONTENTS

Composable Event Streaming Representation (CESR)

Introduction

Status of This Memo

Copyright Notice

Terms of Use

Scope

Normative references

Terms and Definitions

Composability and Domain Representations

Composability

Abstract Domain representations

Transformations between Domains

Examples of circuits of transformations

Example 1

Example 2

Concatenation composability property

Concrete Domain representations

Conversions

Stable Framing Codes in the text domain

Stable type encoding

Stable value encoding

Code characters and lead bytes

Multiple code table approach

Text Coding Scheme Design

Text Code Size
Example of pad size computation
Pre-padding before conversion
Pre-padding a one-byte value
Pre-padding a two-byte value
Pre-padding a three-byte value
Examples of pre-padding
Count or Group Framing Codes
Interleaved non-CESR serializations
Cold start Stream parsing problem
Performant resynchronization with unique start bits
Top-level Stream Starting Tritets
Stream parsing rules
Compact fixed-size codes
Code table selectors
Table types
Tables of Codes for Fixed-length Raw Sizes
Small fixed-length raw-size tables
One-character fixed-length raw-size table
Two-character fixed-length raw size table
Large fixed-length raw-size tables
Large fixed-length raw-size table with 0 Lead Bytes
Large fixed-length raw-size table with 1 lead byte
Large fixed-length raw-size table with 2 lead bytes
Tables for Codes with Variable-length Raw-sizes
Small variable-length raw-size tables

Small variable-length raw-size table with 0 lead bytes
Small variable-length raw-size table with 1 lead byte
Small variable-length raw-size table with 2 lead bytes
Large Variable-length Raw-size Tables
Large variable-length raw-size table with 0 lead bytes
Large variable-length raw-size table with 1 lead byte
Large variable-length raw-size table with 2 lead bytes
Count Code tables
Small Count Code table
Large Count Code table
Protocol genus/version table
Protocol genus/version codes
OpCode tables
Op Code table
Summary of Selector code tables and encoding scheme design
Encoding scheme table
Encoding Scheme Table
Encoding scheme symbols
Encoding Scheme Symbols Table
Special context-specific code tables
Indexed codes
Indexed code table
Encoding scheme format symbol table
Parsing via table design
Table 1
Table 2

Table 3
Table 4
Annex A
Code table entry policy
Table format
Universal Code tables
Universal Code table genus/version codes
Universal Code table genus/version codes that allow genus/…
Universal Code table genus/version codes that do not allow …
KERI/ACDC Protocol Stack Tables
KERI/ACDC protocol genus version table
Master code table for genus/version -_AAACAA (KERI/ACD…
Indexed code table for genus/version --AAACAA (KERI/ACD…
Examples
Version String field
Version 2.XX string field format
Legacy Version 1.XX string field format
Self-addressing identifier (SAID)
Generation and Verification Protocols
Example Computation
Serialization Generation
Order-Preserving Data Structures
Example Python dict to JSON Serialization with SAID
Example Schema Immutability using JSON Schema with SA…
Discussion
Self-addressing Data (SAD) Path Signatures

Streamable SADs
Nested Partial Signatures
Transposable Signature Attachments
SAD Path Language
Description and Usage
CESR Encoding for SAD Path Language
SAD Path Examples
Alternative Pathing / Query Languages
Post-Quantum Security
Bibliography
Normative section
Informative section
Issues
Settings

Composable Event Streaming Representation (CESR)

Specification Status: v1.1

DOI

https://doi.org/10.5281/zenodo.18879945

Latest Draft:

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

Authors:

- Samuel Smith, Prosapien

- Philip Feairheller, HealthKERI

Editors:

- Kevin Griffin, GLEIF

Contributors:

- Samuel Smith, Prosapien
- Philip Feairheller, HealthKERI
- Kevin Griffin, GLEIF
- Nuttawut Kongsuwan
- Trent Larson
- Kent Bull
- Charles Lanahan
- Henk van Cann, Blockchainbird
- Kor Dwarshuis, Blockchainbird

Participate:

GitHub repo

Commit history

Introduction

The Composable Event Streaming Representation (CESR) is a dual text-binary encoding format that has the unique property of text-binary concatenation composability. This Composability property enables the round-trip conversion en-masse of concatenated Primitives between the text domain and binary domain while maintaining the separability of individual Primitives. This enables convenient usability in the text domain and compact transmission in the binary domain. CESR Primitives are self-framing. CESR supports self-framing Group Codes that enable stream processing and pipelining in both the text and binary domains. CESR supports composable text-binary encodings for general data types as well as suites of cryptographic material. Popular cryptographic material suites have compact encodings for efficiency, while less compact encodings provide sufficient extensibility to support all foreseeable types. CESR streams also support interleaved JSON, CBOR, and MGPK serializations. CESR is a universal encoding that uniquely provides dual text and binary domain representations via composable conversion. The CESR protocol is used by other protocols such as [1].

One way to better secure Internet communications is to use cryptographically verifiable Primitives and data structures inside Messages and in support of messaging protocols. Cryptographically verifiable Primitives provide essential building blocks for zero-trust computing and networking architectures. Traditionally, Cryptographic Primitives, including but not limited to digests, salts, seeds (private keys), public keys, and digital signatures,

have been largely represented in some binary encoding. This limits their usability in domains or protocols that are human-centric or equivalently that only support ASCII text-printable characters [RFC20]. These domains include source code, documents, system logs, audit logs, legally defensible archives, Ricardian contracts, and human-readable text documents of many types [RFC4627].

Generic binary-to-text, [12], or simply textual encodings such as Base64 [RFC4648], do not provide any information about the type or size of the underlying Cryptographic Primitive. Base64 only provides “value” information. More recently, [10] was developed as a fit-for-purpose textual encoding of Cryptographic Primitives for shared distributed ledger applications that, in addition to value, may include information about the type and, in some cases, the size of the underlying Cryptographic Primitive [11]. Each application, however, may use a non-interoperable type and optionally size encoding because a binary encoding may include as a subset some codes that are in the text-printable compatible subset of [2] such as ISO Latin-1, [14] or UTF-8 [13]. Interestingly, for a given Cryptographic Primitive, a text-printable type code from a binary code table could be found serendipitously from a set of binary encodings. This is the case for the Multicodec encodings, which are binary but include a subset of “serendipitous” ASCII codes. [8] [7] IPFS. Indeed, some [10] applications take advantage of the binary MultiCodec tables but only use serendipitous text-compatible type codes. Serendipitous text encodings in binary code tables do not generally work for any size or type. So, the serendipitous approach is not universally applicable and is no substitute for a true textual encoding protocol for Cryptographic Primitives.

A textual or binary encoding that includes type, size, and value is Self-Framing. A self-framing Primitive may be extracted from a stream of characters or bytes without needing any additional delimiting characters. Thus, a stream of concatenated Primitives may be parsed without the need to encapsulate each Primitive inside a set of delimiters or an envelope. A textual Self-Framing encoding provides the core capability for a streaming text protocol like [15] or [16]. Although a first-class textual encoding of Cryptographic Primitives is the primary motivation for the CESR protocol, CESR is sufficiently flexible and extensible to support other useful data types, such as integers of various sizes, floating-point numbers, date-times as well as generic text. Thus, the CESR protocol is generally useful to encode data structures of all types into text, not merely those that contain Cryptographic Primitives.

Textual encodings have numerous usability advantages over binary encodings. The one advantage, however, of a binary encoding over text is compactness. An encoding protocol with the property called text-binary concatenation composability or, more succinctly, Composability enables both text’s usability and binary’s compactness. Composability may be the most uniquely innovative and useful feature of the CESR encoding protocol.

No standard text-based encoding protocol provides universal type, size, and value encoding for Cryptographic Primitives as compact atomic values. Providing this capability is one of the primary motivations for the CESR encoding protocol. But text-based atomic cryptographic primitives alone are insufficient for cryptography-heavy protocols. Grouping those primitives into cryptographically verifiable data structures, including messages with attachments, is also essential. Consequently, CESR provides encodings for groups or collections of primitives such as lists, field maps, fixed field data structures, messages, attachments to messages, and arbitrary collections of groups.

Like primitives, CESR group encodings are Self-Framing. This enables efficient stream processing of CESR streams. A CESR parser can efficiently extract whole groups from the stream without parsing into the group. The extracted groups can then be diverted to other processor resources to be processed in parallel. This enables pipelining of CESR streams and messages within a stream.

The support for efficient stream processing is reflected in how a cryptographic commitment to some data is associated with that data. For example, a serialized data structure that constitutes a message may be signed digitally. The signature constitutes a non-repudiable commitment by the holder of the private key to the message. Cryptographically, the signature (commitment) can not be part of the data it signs (commits to). Therefore, the signature must be attached to the message in some way. This constraint also applies to other commitments like cryptographic digests (hashes). The signature may be used as a strong authentication factor for the message. A stream processor may want to drop any messages whose signatures do not verify. One common way of associating commitments to a message is to create a new message that acts as a wrapper or envelope on the original message. The wrapper message includes both the original message and the commitment. However, enveloping or wrapping may defeat efficient stream processing, especially when that envelope is block delimited. The parser now has to parse into the wrapper to find the signature to verify it against the message. The wrapper is discarded. A more stream-processing-friendly approach is to attach commitments to messages as Self-Framing stream parts without creating disposable wrappers. Consequently, CESR provides Self-Framing group encodings for attachments instead of wrappers. Properly, in CESR parlance, a full Message consists of a Message Body plus Attachments.

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-cesr-specification .

Copyright Notice

Terms of Use

Scope

Implementation design of a protocol-based data serialization specification that supports loss-less round-tripping between text and binary representations of compositions of primitives and groups of primitives. The encoding scheme includes first-class cryptographically agile encodings for the full range of cryptographic primitives such as random numbers, digests, secrets, private keys, public keys, signatures, encrypted data, etc. This enables cryptographic heavy protocols to succinctly represent integrated cryptographic primitives in the text domain for improved usability and readability while supporting loss-less round

trip convertibility to binary for more compact transmission and storage. This better supports the increasing demand for cryptographic heavy protocols for enhanced security. Also supported are interleaved JSON, CBOR, and MGPK encodings of field maps that contain cryptographic primitives as field values. The application scope includes any electronically transmitted information. The implementation dependency scope includes Base64 encoding/decoding libraries, standardized cryptographic primitive definitions, JSON, CBOR, and MGPK libraries.

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/

Attachment (Attachments, *attachment*)

A serialized data structure that supplements a Message Body, typically following it in transmission. Attachments may include signatures, receipts, or other metadata necessary to verify or interpret the message. They are considered an integral part of the overall Message .

More: CESR also provides the attachment codes needed for differentiating the types of cryptographic material (such as signatures) used as attachments on all event types for the KERI.

Autonomic identifier (*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

CESR (*composable-event-streaming-representation*)

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

Also called ‘CESR’. This compact encoding scheme fully supports both textual and binary streaming applications of attached crypto material of all types. This approach includes composability in both the textual and binary streaming domains. The primitive may be the minimum possible but still composable size.

Making composability a guaranteed property allows future extensible support of new compositions of streaming formats based on pre-existing core primitives and compositions of core primitives. This enables optimized stream processing in both the binary and text domains.

More in extended KERI glossary

CESRgram

a CESR-encoded message — essentially a serialized binary or text representation of an event, message, or attachment using the CESR format.

Composability (Concatenation Composability, *composability*)

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

short for text-binary concatenation composability. An encoding has Composability when any set of Self-Framing concatenated Primitives expressed in either the Text domain or Binary domain may be converted as a group to the other Domain and back again without loss.

Source: Dr. S.Smith

More in extended KERI glossary

Cryptographic agility (agility, *cryptographic-agility*)

a SAID MUST include a pre-pended derivation code that specifies the cryptographic algorithm used to generate the digest. This provides cryptographic agility for CESR.

Cryptographic primitive (*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

Domain (*domain*)

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

a representation of a primitive either Text (T), Binary (B) or Raw binary (R).

Source: Dr. S. Smith

Beware: outside of CESR but within the internet world, the term ‘domain’ mostly refers to the concept of a *domain name*.

More in extended KERI glossary

Framing code (*framing-code*)

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

a code that delineates a number of characters or bytes, as appropriate, that can be extracted atomically from a stream .

Source: Dr. S. Smith

Group/Count codes (*group-count-codes*)

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

special Framing Codes that can be specified to support groups of Primitives which make them pipelinable. Self-framing grouping using Count Codes is one of the primary advantages of composable encoding.

Source: Dr. S.Smith

IPFS (*interplanetary-file-system*)

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

The InterPlanetary File System (IPFS) is a protocol, hypermedia and file sharing peer-to-peer network for sharing data using a distributed hash table to store provider information.

Source wikipedia

JSONPath

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

In computer software, JSONPath is a query language for querying values in JSON.

See more: Wikipedia JSONPath

⇨ **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

⇨ **Message** (Messages, *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.

CESR-based messages typically can have Attachments and their attachment codes.

Mid-padding (*mid-padding*)

the chosen approach for CESR. Adds leading pad bytes to the value portion pre-conversion, effectively placing the padding after the framing code but before the value i.e. mid-padding.

Source: dr. S. Smith

Ondex (*ondex*)

Ondex is “other index”. Some indexed signature codes have two indices, so for those that do its the other one.

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

⇨ **Quadlet** (*quadlet*)

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

a group of 4 characters in the T domain and equivalently in triplets of 3 bytes each in the B domain used to define variable size.

Source: Dr. S. Smith

More in extended KERI glossary

⇨ **Self-framing** (*self-framing*)

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

a textual or binary encoding that begins with type, size, and value so that a parser knows how many characters (when textual) or bytes (when binary) to extract from the stream for a given element without parsing the rest of the characters or bytes in the element is Self-Framing.

More in extended KERI glossary

⇨ **Stable** (*stable*)

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

Refers to the state of cryptographic verifiability across a network or system. It generally implies that a particular identifier, event, or data set is consistent, fully verified, and cannot be contested within KERI.

More in extended KERI glossary

⇨ **Stream** (*stream*)

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

a CESR Stream is any set of concatenated Primitives, concatenated groups of Primitives, or hierarchically composed groups of primitives.

Source: Dr. S. Smith

More in extended KERI glossary

Tritet (*tritet*)

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

3 bits. See Performant resynchronization with unique start bits.

Source: Dr. S. Smith

Variable Length (*variable-length*)

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

a type of count code allowing for vaiable size signatures or attachments which can be parsed to get the full size.

Source: Dr. S. Smith

⇨ **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

⇨ **Version String** (*version-string*)

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

the first field in any top-level KERI field map in which it appears.

More in extended KERI glossary

Composability and Domain Representations

Composability

An encoding has Composability when any set of Self-Framing concatenated Primitives expressed in either the Text domain or Binary domain may be converted as a group to the other Domain and back again without loss. Essentially, Composability provides round-trippable lossless conversion between Text and Binary domain representations of any set of concatenated Primitives when converted as a set not merely individually. The property enables a Stream processor to safely convert en-masse a Stream in the Text domain to an equivalent Stream in the Binary domain for compact transmission that may be safely converted back to Text domain en-masse by a Stream processor at the other end for further processing or archival storage. All compliant encoded Primitives MUST be Composable. All compliant encoded Primitives MUST be self-framing.

The use of Count Codes as independently composable groups enables hierarchical compositions. Such a hierarchically composable encoding protocol enables pipelining (multiplexing and de-multiplexing) of complex Streams in either text or compact binary. This allows management at scale for high-bandwidth applications that benefit from core affinity off-loading of Streams [17]. All Count Code groups of Primitives or other compositions of Primitives and Count code groups MUST be Composable. All Count Code groups of Primitives or other compositions of Primitives and Count code groups MUST be self-framing.

Abstract Domain representations

The Cryptographic Primitives defined in CESR inhabit three different Domains each with a different representation. The first Domain is called Text for streamable text and is denoted as ‘T’. The second Domain is called Binary for streamable binary and is denoted as ‘B’. Composability is defined between the ‘T’ and ‘B’ domains. The third Domain is called Raw, which is non-streamable binary, and is denoted as ‘R’. The third Domain is special because Primitives in this Domain are represented by a pair or two-tuple of values namely (text code, raw binary) or `(code, raw)` for short. The text code element of the ‘R’ domain pair is a string of one or more text characters that provides the type and size information for the encoded Primitive when in the ‘T’ domain. The raw binary element is composed of bytes. The actual use of Cryptographic Primitives happens in the ‘R’ domain using the raw binary element of the `(code, raw)` pair. Cryptographic Primitive values are usually represented as strings of bytes that represent very large integers. Cryptographic libraries typically assume that the inputs and outputs of their functions will be such strings of bytes. The raw binary element of the ‘R’ domain pair is such a string of bytes. The CESR protocol, however, is not limited to merely encoding Cryptographic Primitives but any primary data type (numbers, text, datetimes, lists, maps) may be encoded in a composable way.

A given Primitive in the ‘T’ domain is denoted with `t`. A member of an indexed set of Primitives in the ‘T’ domain is denoted with `t[k]`; where k is the member. Likewise, a given Primitive in the ‘B’ domain is denoted with `b`. A member of an indexed set of Primitives in the ‘B’ domain is denoted with `b[k]`. Similarly, a given Primitive in the ‘R’ domain is denoted with `r`. A member of an indexed set of Primitives in the R’ domain is denoted with `r[k]`.

Transformations between Domains

Although the Composability property mentioned in the previous section only applies to conversions back and forth between the ‘T’, and ‘B’, domains, conversions between the ‘R’, and ‘T’ domains, as well as conversions between the ‘R’ and ‘B’ domains, also are defined and supported by the protocol as described in detail in this section. As a result, there is a total of six transformations, one in each direction, among the three Domains.

Let `T(B)` denote the abstract transformation function from the ‘B’ domain to the ‘T’ domain. This is the dual of `B(T)` below.

Let `B(T)` denote the abstract transformation function from the ‘T’ domain to the ‘B’ domain. This is the dual of `T(B)` above.

Let `T(R)` denote the abstract transformation function from the ‘R’ domain to the ‘T’ domain. This is the dual of `R(T)` below.

Let `R(T)` denote the abstract transformation function from the ‘T’ domain to the ‘R’ domain. This is the dual of `T(R)` above.

Let `B(R)` denote the abstract transformation function from the ‘R’ domain to the ‘B’ domain. This is the dual of `R(B)` below.

Let `R(B)` denote the abstract transformation function from the ‘B’ domain to the ‘R’ domain. This is the dual of `B(R)` above.

Given these transformations, a complete a circuit of transformations can be completed that starts in any of the three Domains and then crosses over the other two Domains in either direction. All compliant implementations MUST support the transformations between all three domains.

Examples of circuits of transformations

Example 1

Starting in the ‘R’ domain, a circuit that crosses into the ‘T’ and ‘B’ domains can be traversed and then crossed back into the ‘R’ domain as follows:

```
R->T(R)->T->B(T)->B->R(B)->R
```

Example 2

Likewise, starting in the ‘R’ domain, a circuit that crosses into the ‘B’ and ‘T’ domains and then crossed back into the ‘R’ domain as follows:

```
R->B(R)->B->T(B)->T->R(T)->R
```

Concatenation composability property

Let `+` represent concatenation. Concatenation is associative and may be applied to any two Primitives or any two groups or sets of concatenated Primitives. For example:

```
t[0] + t[1] + t[2] + t[3] = (t[0] + t[1]) + (t[2] + t[3])
```

If we let `cat(x[k])` denote the concatenation of all elements of a set of indexed Primitives `x[k]` where each element is indexed by a unique value of `k`. Given the indexed representation, the transformation can be expressed between Domains of a concatenated set of Primitives as follows:

Let `T(cat(b[k]))` denote the concrete transformation of a given concatenated set of primitives, `cat(b[k])` from the B domain to the T domain.

Let `B(cat(t[k]))` denote the concrete transformation of a given concatenated set of Primitives, `cat(t[k])` from the T domain to the B domain.

The Concatenation Composability property or Composability for short, between T and B is expressed as follows:

Given a set of Primitives `b[k]` and `t[k]` and transformations `T(B)` and `B(T)` such that `t[k] = T(b[k])` and `b[k] = B(t[k])` for all `k`, then `T(B)` and `B(T)` are jointly concatenation composable if and only if,

```
T(cat(b[k]))=cat(T(b[k])) and B(cat(t[k]))=cat(B(t[k])) for all
k.
```

Basically, Composability (over concatenation) means that the transformation of a set (as a whole) of concatenated Primitives is equal to the concatenation of the set of individually transformed Primitives. Each and every Primitive or Count Code group of primitives MUST satisfy the Concatenation Composability property.

For example, suppose there are two Primitives in the Text domain, namely, `t[0]` and `t[1]` that each transforms, respectively, to primitives in the Binary domain, namely, `b[0]` and `b[1]`. The transformation duals, `B(T)` and `T(B)`, are composable if and only if,

```
B(t[0] + t[1]) = B(t[0]) + B(t[1]) = b[0] + b[1]
```

and

```
T(b[0] + b[1]) = T(b[0]) + T(b[1]) = t[0] + t[1].
```

The Composability property defined above allows us to create arbitrary compositions of primitives via concatenation in either the T or B domain and then convert the composition en masse to the other domain and then de-concatenate the result without loss. The self-framing property of the primitives enables de-concatenation.

The Composability property is an essential building block for streaming in either Domain. The use of framing Primitives that count or group other Primitives enables multiplexing and demultiplexing of arbitrary groups of Primitives for pipelining and/or on or offloading of streams. The Text domain representation of a Stream enables better usability (readability), and the Binary domain representation of a Stream enables better compactness. In addition, pipelined hierarchical composition codes allow efficient conversion or offloading for concurrent processing of composed (concatenated) groups of Primitives in a Stream without having to individually parse each Primitive before off-loading.

Concrete Domain representations

The Text, 'T', domain representations in CESR MUST use only the characters from the URL/filename safe variant of the IETF RFC-4648 Base64 standard[RFC4648]. Unless otherwise indicated, all references to Base64 [RFC4648] in this document imply the URL and filename safe variant. The URL and filename safe variant of Base64 uses in order the 64 characters `A to Z`, `a to z`, `0 to 9`, `-`, and `_` to encode 6 bits of information. In addition, Base64 uses the `=` character for padding, but CESR does not use the `=` character for any purpose because all CESR-encoded Primitives are composable.

It is notable that Base64 [RFC4648] by itself does not satisfy the Composability property and as a result, employs pad characters to ensure one-way convertibility between the Binary domain and the Text domain.

In CESR, however, both 'T' and 'B' domain representations include a prepended Framing Code prefix that is structured to ensure Composition.

Suppose that Base64 characters are used in the Text domain and binary bytes are used in the Binary domain, called respectively, naive text and naive binary encodings and Domains. Recall that a byte encodes 8 bits of information and a Base64 character encodes 6 bits of information. Furthermore, suppose that there are three Primitives denoted `a`, `b`, and `c` in the naive Binary domain with lengths of 1, 2, and 3 bytes, respectively.

In the following diagrams, each byte is denoted in a naive binary Primitive with zero-based most significant bit first indices, e.g., `a1` is bit one from `a`, `a0` is bit zero, and `A0` for byte zero, `A1` for byte 1, etc.

The byte and bit-level diagrams for `a` are shown below, where `A` is used to denote its bytes:

```
|           A0          |
|a7:a6:a5:a4:a3:a2:a1:a0|
```

Likewise, for `b` below:

```
|           B1          |           B0          |
|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|
```

And finally, for `c` below:

```
|           C2          |           C1          |           C0
|
|c7:c6:c5:c4:c3:c2:c1:c0|c7:c6:c5:c4:c3:c2:c1:c0|c7:c6:c5:c4:c3:c
2:c1:c0|
```

Conversions

Given that the constraint of alignment on 24-bit boundaries in either the Text domain or the Binary domain needs to be satisfied to reach Composibility, this paragraph explains how it's done.

When doing a naive Base64 conversion of a naive binary Primitive, one Base64 character represents only six bits from a given byte. In the following diagrams, each character of a Base64 conversion is denoted using zero-based indices, with the most significant character first.

Therefore, encoding `a` in Base64 requires at least two Base64 characters because the zeroth character only captures the six bits from the first byte, and another character is needed to capture the other two bits. The convention in Base64 uses a Base64 character where the non-coding bits are zeros. This is diagrammed as follows:

```
|           A0          |
|a7:a6:a5:a4:a3:a2:a1:a0|z3:z2:z1:z0|
|        T1       |        T0       |
```

where `aX` represents a bit from `a`, `AX` represents a byte from `a`, `zX` represents a zeroed pad bit, and `TX` represents a non-pad character from the converted Base64 text representing one hextet of information from the converted binary string.

Naive Base64 encoding always pads each individual conversion of a string of bytes to an even multiple of four characters. This provides a property that is not true Composability but does ensure that multiple distinct concatenated conversions from binary to Base64

text are separable. It may be described as a sort of one-way composability. So, with pad characters, denoted by replacing the spaces with `=` characters, the Base64 conversion of `a` is as follows:

```
|           A0          |
|a7:a6:a5:a4:a3:a2:a1:a0|z3:z2:z1:z0|
|        T3       |        T2       |========P1=======|========P0
=======|
```

where `PX` represents a trailing pad character. We see that Base64 conversion effectively left shifts `a` by four bits plus two pad characters. In other words, the Base64 conversion of `a` is no longer right-aligned with respect to the trailing Base64 character.

Likewise, `b` requires at least three Base64 characters to capture all of its sixteen bits of information as follows:

```
|           B1          |           B0          |
|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|z1:z0|
|        T2       |        T1       |        T0       |
```

where `bX` represents a bit from `b`, `BX` represents a byte from `b`, `zX` represents a zeroed pad bit, and `TX` represents a non-pad character from the converted Base64 text representing one hextet of information from the converted binary string.

Alignment on a four-character (24-bit) boundary requires one pad character this becomes:

```
|           B1          |           B0          |
|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|z1:z0|
|        T2       |        T1       |        T0       |========P0
=======|
```

where `PX` represents a trailing pad character. We see that Base64 conversion effectively left shifts `b` by two bits plus one pad character. In other words, the Base64 conversion of `b` is no longer right-aligned with respect to the trailing Base64 character.

Finally, `c` requires exactly four Base64 characters to capture all of its twenty-four bits of information. There are no pad characters required.

```
|           C0          |           C1          |           C2
|
|c7:c6:c5:c4:c3:c2:c1:c0|c7:c6:c5:c4:c3:c2:c1:c0|c7:c6:c5:c4:c3:c
2:c1:c0|
```

```
|        T3       |        T2       |        T1       |        T0       |
|
```

where `cX` represents a bit from `c`, `CX` represents a byte from `c`, and `TX` represents a non-pad character from the converted Base64 text representing one hextet of information from the converted binary string. There are no bit shifts because there are no pad bits nor pad characters needed, and the resulting Base64 conversion is right aligned with respect to the trailing Base64 character.

Suppose `a + b` now is concatenated into a three-byte composition in the naive Binary domain before Base64 encoding the concatenated whole.

```
|           A0          |           B1          |           B0          |
|a7:a6:a5:a4:a3:a2:a1:a0|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|
|        T3       |        T2       |        T1       |        T0       |
|
```

The least significant two bits of `A0` are encoded into the same character, `T2`, as the most significant four bits of `B1`. Therefore, a Text domain parser would be unable to cleanly de-concatenate on a character-by-character basis the conversion of `a + b` into separate Text domain Primitives. Therefore, standard (naive) binary to Base64 conversion does not satisfy the Composability constraint.

Starting instead in the Text domain with Primitives `u` and `v` of lengths 1 and 3 characters, respectively, these two Primitives can be concatenated as `u + v` in the Text domain and then converted as a whole to naive binary.

```
|        U0       |        V2       |        V1       |        V0       |
|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|
|           B2          |           B1          |           B0          |
|
```

All six bits of information in `U0` are included in `B2` along with the least significant two bits of information in `V2`. Therefore, a Binary domain parser is unable to cleanly de-concatenate on a byte-by-byte basis the conversion of `u + v` into separate Binary domain Primitives. Therefore, standard (naive) Base64 to binary conversion does not satisfy the Composability constraint.

The Composability property is satisfied only if each Primitive in the ‘T’ domain is an integer multiple of four Base64 characters (24 bits) and each Primitive in the ‘B’ domain is an integer multiple of three bytes (24 bits). Each of either four Base64 text characters or

three binary bytes captures 24 bits of information. Twenty-four is the least common multiple of six and eight. Therefore, in order to cleanly capture integer multiples of twenty-four bits of information, Primitive lengths MUST be integer multiples of either four Base64 text characters or three binary bytes in their respective Domains. Given that the constraint of alignment on 24-bit boundaries in either Text domain or Binary domain is satisfied, the conversion of concatenated Primitives in one Domain never results in the same byte or character in the converted Domain sharing bits from two adjacent Primitives. This constraint of 24-bit alignment, therefore, satisfies the Composability property.

To elaborate, when converting Streams made up of concatenated Primitives back and forth between the ‘T’ and ‘B’ domains, the converted results will not align on byte or character boundaries at the end of each Primitive unless the Primitives themselves are integer multiples of twenty-four bits of information. In other words, all Primitives MUST be aligned on 24-bit boundaries to satisfy the Composability property. This means that the length of any Primitive in the ‘B’ domain MUST be an integer multiple of three binary bytes with a minimum length of three binary bytes. Likewise, this means that the length of any Primitive in the ‘T’ domain MUST be an integer multiple of four Base64 characters with a minimum length of four Base64 characters.

Stable Framing Codes in the text domain

There are many coding schemes that could satisfy the Composability constraint of alignment on 24-bit boundaries. A primary design goal of CESR is to select an encoding approach that provides high usability, readability, or human friendliness in the ‘T’ domain. This type of usability goal simply is not realizable in the ‘B’ domain. The ‘B’ domain’s purpose is merely to provide convenient compactness at scale. Usability in the ‘T’ domain is maximized when the type portion of the prepended Framing Code and its postpended value are Stable, i.e., ‘invariant’. The type portion of all compliant prepended Framing Codes MUST be stable in the Text domain. Stable type encoding is defined in the following section.

Stable type encoding

Stable type coding makes it much easier to recognize Primitives of a given type when debugging source, reading Messages, or documents in the ‘T’ domain that include encoded Primitives. This is true even when those Primitives have different lengths or values. For Primitive types with fixed lengths, i.e., all Primitives of that type have the same length, Stable type coding aids visual type and visual size recognition. Stable type coding means that the leading characters that determine the type do not change when any other portion of the primitive changes.

The usability of Stable type coding is maximized when the type portion appears first in the Framing Code. Stability also requires that for a given type, the type coding portion MUST consume a fixed integral number of characters in the ‘T’ domain. To clarify, as used here, Stable type coding in the ‘T’ domain never shares information bits with either

length or value coding in any given Framing Code character and appears first in the Framing Code. Stable type coding in the ‘T’ domain translates to Stable type coding in the ‘B’ domain, except that the type coding portion of the Framing Code MAY or MAY NOT respect byte boundaries. This is an acceptable tradeoff because binary-domain parsing tools easily accommodate bit fields and bit shifts while text-domain parsing tools do not. Generally, Text domain parsing tools only process whole characters. This is another reason to impose a stability constraint on the ‘T’ domain type coding instead of the ‘B’ domain.

Therefore, the type portion MUST begin the Framing Code, and the type coding portion MUST consume a fixed integral number of characters in the ‘T’ domain.

Stable value encoding

A secondary usability constraint is recognizable or readable Stable value coding in the Text, ‘T’, domain. Stable value encoding means that the trailing Base64 characters that encode the primitive value are right aligned. This means one can manually confirm values are the same. Not all Primitives benefit from Stable value coding. Any representation of a value that is a long random string of characters is essentially unreadable or recognizable versus some other representation. Consequently, bit shifts of the value that result in leading or trailing zero pads, as long as they are static, do not change the readability. This is not true, however, of values that are small numbers. Base64 encodings of small numbers are readable. for example, the numerical sequence of decimal numbers, `0, 1, 2`, is recognizable as the sequence of Base64 characters, `A, B, C`. Thus, all else equal, readable Stable value encodings also contribute to usability, at least in some cases. The combination of Stable leading type encoding and Stable trailing value encoding means that any zero padding MUST appear in the middle of the Primitive, after the type code, but before the value.

Therefore the value portion of any primitive MUST be right aligned.

Code characters and lead bytes

Two ways exist to provide the required alignment on 24-bit boundaries to satisfy the Composability property defined above. One is to post-pad (with trailing pad characters, `=`) the Text domain encoding to ensure that the ‘T’ domain Primitive has a total size (length) of an integer multiple of 4. This is what naive Base64 encoding does. The other way is to pre-pad leading bytes of zeros to the raw binary value portion before conversion to Base64 to ensure the total size of the raw binary value with pre-pad bytes is an integer multiple of 3 bytes. This ensures that the size in characters of the Base64 conversion of the pre-padded raw binary is an integer multiple of 4 characters. This means that, effectively, value padding shows up as a mid-pad relative to the full Primitive with a prepended type code.

Given pre-padded values, one of two options is available that depends on the specific code. In the first option, an appropriate number of text characters that result from the conversion of a porting of the leading pre-pad zero bytes are replaced with the appropriate number of code characters. In the second option, the code characters are pre-pended to the conversion with leading zeros intact. In the second option, the length of the pre-pended type code is also an integer multiple of 4 characters. In either option, the total length of the ‘T’ domain Primitive with code is an integer multiple of 4 characters.

The first option may be more compact in some cases than the second. The second option may be easier to compute in some cases. The most significant advantage of the second option is that the value portion is Stable and more readable both in the Text, ‘T’, domain and in the Binary, ‘B’, domain because the value portion is not shifted by the Base64 conversion as it is with the first option.

In order to avoid confusion with the use of the term ‘pad character’ when pre-padding with bytes that are not replaced later, the term ‘lead bytes’ is used. The term pad may be confusing not merely because both ways use a type of padding, but it is also true that the number of pad characters when padding post-conversion equals the number of lead bytes when padding pre-conversion.

Suppose that the raw binary value is 32 bytes in length. The next higher integer multiple of 3 is 33 bytes. Thus 1 additional leading pad byte is needed to make the size (length in byte) of raw binary an integer multiple of 3. The 1 lead byte makes that combination a total of 33 bytes in length. The resultant Base64 converted value will be 44 characters in length, which is an integer multiple of 4 characters. In contrast, recall that when a 32-byte raw binary value is converted to Base64, the converted value will have 1 trailing pad character. In both cases, the resultant length in Base64 is 44 characters.

Similarly, a 64-byte raw binary value needs 2 lead bytes to make the combination 66 bytes in length, where 66 is the next integer multiple of 3 greater than 64. When converted, the result is 88 characters in length. The number of pad characters added on the result of the Base64 conversion of a 64-byte raw binary is also 2.

In summary, there are two possibilities for CESR’s coding scheme to ensure a composable 24-bit alignment. The first is to add trailing pad characters post-conversion. The second is to add leading pad bytes to the value portion pre-conversion, effectively placing the padding after the framing code but before the value i.e. mid-padding. Because of the greater readability of the value portion of both the fully qualified Text, ‘T’, or fully qualified Binary, ‘B’, domain representations, the second approach was chosen for CESR.

Therefore all CESR primitives MUST employ mid-padding as defined.

Multiple code table approach

The design goals for CESR Framing Codes include minimizing the Framing Code size for the most frequently used (most popular) codes while also supporting a sufficiently comprehensive set of codes for all foreseeable current and future applications. This requires a high degree of both flexibility and extensibility. This is best achieved with multiple code tables with a different coding scheme optimized for a different set of features instead of a single one-size-fits-all scheme. A specification that supports multiple coding schemes may appear on the surface to be much more complex to implement, but careful design of the coding schemes can reduce implementation complexity by using a relatively simple single integrated parse and conversion table. Parsing in any given Domain given Stable type codes may then be implemented with a single function that simply reads the appropriate type selector in the table to know how to parse and convert the rest of the Primitive.

Each code table MUST be uniquely indicated by the first character of the type code in the 'T' domain.

Text Coding Scheme Design

Text Code Size

Recall that the 'R' domain representation is a pair `(text code, raw binary)`. The text code is Stable and begins with one or more Base64 characters that provide the Primitive type and may also include one or more additional characters that provide the length. The actual usable cryptographic material is provided by the raw binary element.

The corresponding 'T' domain representation of this pair is created by first prepending leading pad bytes of zeros to the raw binary element. This result is then converted to Base64. Depending on the code, either the frontmost characters that result from the Base64 conversion of leading pad bytes of zeros are replaced with the text code element of appropriate size in characters, or an appropriately sized text code element is prepended to the conversion without replacing any characters.

Recall that when the length of a given naive binary string is not an integer multiple of three bytes, standard Base64 conversion software appends one or two pad characters to the resultant Base64 conversion.

With standard Base64 conversion that employs pad characters, the Text domain representation that results from the individual conversion of a set of binary strings when concatenated in the Text domain after conversion and stripping off pad characters is not necessarily equivalent to the Text domain representation that results from converting en-masse to text the concatenation of the same set of binary strings and then stripping off pad characters. In the latter case, knowledge of the set of binary strings is lost because the resultant conversion may have bits from two binary bytes concatenated in one text

character. Restated, the problem with standard Base64 is that it does not preserve byte boundaries after the en-masse conversion of concatenated binary strings. Consequently, standard (naive) Base64 does not provide two-way or true Composability as defined above.

To elaborate, the number of pad characters appended with standard Base64 encoding is a function of the length of the binary string. Let `N` be the length in bytes of the binary string. When `N mod 3 = 1`, then there are 8 bits in the remainder that are encoded into Base64. Recall from the examples above that a single byte (8 bits) requires two Base64 characters. The first encodes 6 bits, and the second encodes the remaining 2 bits for a total of 8 bits. The last character is selected such that its non-coding 4 bits are zero. Thus, two additional pad characters are required to pad out the resulting conversion so that its length is an integer multiple of 4 Base64 characters. Furthermore, when `N mod 3 = 1`, then 2 more zeroed bytes are added to the length of the binary string such that `M = N + 2` would result in `M mod 3 = 0` or equivalently `N + 2 mod 3 = 0`.

Similarly, when `N mod 3 = 2`, there are two bytes (16 bits) in the remainder that are encoded into Base64. Recall from the examples above that two bytes (16 bits) require three Base64 characters. The first two encode 6 bits each (for 12 bits), and the third encodes the remaining 4 bits for a total of 16. The last character is selected such that its non-coding 2 bits are zero. Thus, one additional trailing pad character is required to pad out the resulting conversion so that its length is an integer multiple of 4 characters. Furthermore, when `N mod 3 = 2`, then the addition of 1 more byte of zeros added to the length of the binary string such that `M = N + 1` would result in `M mod 3 = 0` or equivalently `N + 2 mod 3 = 0`. Thus, the number of leading pre-pad zeroed bytes needed to align the binary string on a 24-bit boundary is the same as the number of trailing pad characters needed to align the converted Base64 text string on a 24-bit boundary.

Finally, when `N mod 3 = 0`, then the binary string is already aligned on a 24-bit boundary and no trailing pad characters are required to ensure the length of the Base64 conversion is an integer multiple of 4 characters. Likewise, no leading pad bytes are required to ensure the length of the binary string is an integer multiple of 3 bytes.

Thus, in all three cases, the number of trailing post-pad characters, if any, needed to align the converted Base64 text string on a 24-bit boundary is the same as the number of leading pre-pad bytes, if any, needed to align the binary string on a 24-bit boundary.

The number of required trailing Base64 post-pad characters or, equivalently the number of leading pre-pad zeroed bytes to ensure 24-bit alignment may be computed with the following formula:

`ps = (3 - (N mod 3)) mod 3)`, where `ps` is the pad size (pre-pad bytes or post-pad characters) and `N` is the size in bytes of the binary string.

Recall that Composability is provided here by prepending text codes that are of the appropriate length to ensure 24-bit boundaries in both the 'T' and the corresponding 'B' domain. The advantage of this approach is that naive Base64 software tooling may be used to convert back and forth between the 'T' and 'B' domains, i.e., `T(B)` is naive Base64 encode, and `B(T)` is naive Base64 decode. In other words, CESR Primitives are compatible with existing Base64 (RFC-4648) tooling. Whereas new software tooling is needed for conversions between the 'R' and 'T' domains, e.g., `T(R)` and `R(T)` and the 'R' and 'B' domains, e.g., `B(R)` and `R(B)`.

The pad size computation is also useful for computing the size of the text codes. Because true Composability also requires that the 'T' domain value MUST be an integer multiple of 4 characters in length, the size of the text code MUST also be a function of the pad size, `ps`, and hence the length of the raw binary element, `N`. Thus, the size of the text code in Base64 characters MUST be a function of the equivalent pad size determined by the length `N mod 3` of the raw binary value.

Example of pad size computation

Let `M` be a non-negative integer-valued variable then:

Pad Size	Code Size
0	4•M
1	4•M + 1
2	4•M + 2

The minimum code sizes are 4, 1, and 2 characters for pad sizes of 0, 1, and 2 characters with minimum `M` equaling 1, 0, and 0, respectively. By increasing `M`, there are larger code sizes for a given pad size.

Pre-padding before conversion

Returning to the examples above, observe what happens when the binary strings are pre-padded with zeroed leading pad bytes of the appropriate length given by `ps = (3 - (N mod 3)) mod 3)` where `ps` is the number of leading pad bytes and `N` is the length of the raw binary string before padding is prepended.

Pre-padding a one-byte value

For the one-byte raw binary string `a`, `ps` is two. The pre-padded conversion results in the following:

```
|           Z1          |           Z0          |           A0          |
```

```
|z7:z6:z5:z4:z3:z2:z1:z0|z7:z6:z5:z4:z3:z2:z1:z0|a7:a6:a5:a4:a3:a
2:a1:a0|
|        T3       |        T2       |        T1       |        T0
|
```

where `ZX` represents a zeroed pre-pad byte, `zX` represents a zeroed pre-pad bit, `AX` represents a byte from `a`, `aX` represents a bit from `a`, and `TX` represents a Base64 character that results from the Base64 conversion of the pre-padded `a`.

It is noteworthy that the first two (i.e., `ps`) characters of the conversion, namely `T3T2`, do not include any bits of information from `a`. This also means that `T3T2` can be modified after conversion without impacting the appearance or value of the converted `a` that appears solely in `T1T0`, i.e., there is no overlap. Moreover, the resulting Base64 conversion of `a` is right aligned with respect to the trailing Base64 character. This means that the numerical values for `a` from such an unshifted Base64 conversion can be 'read' and understood. This also means that a text-based parser on a character-by-character basis can cleanly process `T3T2` separate from the Base64 encoding of `a` that appears in `T1T0`. Given this separation, `T3T2` can be replaced with two-character Base64 textual type code `S1S0` as follows:

```
|           Z1          |           Z0          |           A0
|
|z7:z6:z5:z4:z3:z2:z1:z0|z7:z6:z5:z4:z3:z2:z1:z0|a7:a6:a5:a4:a3:a
2:a1:a0|
|        S1       |        S0       |        T1       |        T0
|
|s5:s4:s3:s2:s1:s0|s5:s4|s3:s2:s1:s0|z3:z2:z1:z0|a7:a6:a5:a4:a3:a
2:a1:a0|
```

where `ZX` represents a zeroed pre-pad byte, `zX` represents a zeroed pre-pad bit, `AX` represents a byte from `a`, `aX` represents a bit from `a`, `TX` represents a Base64 character that results from the Base64 conversion of the pre-padded `a`, `SX` represents a Base64 code character replacing one of the `TX`, and `sX` is a code bit. The resultant four-character Base64 encoded Primitive would be `S1S0T1T0`.

When `S1S0T1T0` is converted back to binary from Base64, the result would be as follows:

```
|        S1       |        S0       |        T1       |        T0
|
|s5:s4:s3:s2:s1:s0|s5:s4:s3:s2:s1:s0|z3:z2:z1:z0|a7:a6:a5:a4:a3:a
2:a1:a0|
|           U1          |           U0          |           A0
|
```

where `CX` represents a Base64 code character replacing one of the `TX`, `cX` is a code bit, `UX` represents a byte from the converted code char, which may include zeroed bits, `zX` represents a zeroed pre-pad bit, `AX` represents a byte from `a`, `aX` represents a bit from `a`, and `TX` represents a Base64 character that results from the Base64 conversion of the pre-padded `a`.

Stripping off `U1U0` leaves `a` in its original state. It is noteworthy that the code characters (only) are effectively left shifted 4 bits after conversion. The code characters `S1S0` can be recovered as the first two characters that are obtained from simply converting `U1U0` only back to Base64.

Pre-padding a two-byte value

For the two-byte raw binary string `b`, `ps` is one. The resultant four-character Base64 encoded Primitive would be `S0T2T1T0`.

When `S0T2T1T0` is converted back to binary from Base64, the result would be as follows:

```
|        S0       |        T2       |        T1       |        T0
|
|s5:s4:s3:s2:s1:s0|z1:z0|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b
2:b1:b0|
|           U0          |           B1          |           B0
|
```

where `SX` represents a Base64 code character replacing one of the `TX`, `sX` is a code bit, `UX` represents byte from converted code char which may include zeroed bits, `zX` represents a zeroed pre-pad bit, `BX` represents a byte from `b`, `bX` represents a bit from `b`, and `TX` represents a Base64 character that results from the Base64 conversion of the pre-padded `b`.

Stripping off `U0` leaves `b` in its original state. It is noteworthy is that the code character (only) is effectively left shifted 2 bits after conversion. The code character `S0` can be recovered as the first character obtained from simply converting `U0` only to Base64.

Pre-padding a three-byte value

For the three-byte raw binary string `c`, `ps` is zero. So pre-padding is not needed.

```
|        T3       |        T2       |        T1       |        T0
|
|c7:c6:c5:c4:c3:c2:c1:c0|c7:c6:c5:c4:c3:c2:c1:c0|c7:c6:c5:c4:c3:c
2:c1:c0|
|           C2          |           C1          |           C2
|
```

where `cX` represents a bit from `c`, `CX` represents a byte from `c`, and `TX` represents a non-pad character from the converted Base64 text representing one hextet of information from the converted binary string. There are no bit shifts because there are no pad bits nor pad characters needed, and the resulting Base64 conversion is right aligned with respect to the trailing Base64 character.

Without pad characters, however, there is no room to hold a type code. Consequently, any text type code is just prepended to the conversion. The prepended type code MUST be an integer multiple of four Base64 characters. Let `S3S2S1S0` be the type code, then the full Primitive with code and converted raw binary is given by the eight-character Base64 string `S3S2S1S0T3T2T1T0`.

When `S3S2S1S0T3T2T1T0` is converted back to binary, there is no overlap or bit shifting because both the code and raw binary `c` are each separately aligned on twenty-four-bit boundaries.

Examples of pre-padding

Suppose that two-byte raw binary numbers are to be encoded into CESR using the pre-pad approach described above. In order to achieve 24-bit alignment, the pre-pad size for two-byte numbers is 1 byte. As described above, this means the minimally sized text code is 1 Base64 character. Suppose that the text code is `M` (Base64). The following table provides examples of encoding the different two-byte raw binary values in the three Domains: Raw, Text, and Binary. Recall that the Raw domain is expressed by a tuple of (code, raw) where the code is Base64 text and the raw is the raw binary value without code. For readability, raw binary values are represented in hexadecimal notation.

Raw	Text	Binary
(“M”, 0x0000)	“MAAA”	0x300000
(“M”, 0x0001)	“MAAB”	0x300001
(“M”, 0xffff)	“MP__”	0x30ffff

With this approach, both the Binary domain and Text domain representations are as compact as possible for a fully qualified Primitive that satisfies the Composability property. The Text domain representation has a Stable readable code and a Stable readable value. The Binary domain is value right aligned. The Text domain representation consists of 4 text printable characters from the Base64 set of characters, and the Binary domain representation consists of 3 bytes. Both can be parsed in each Domain along character/byte boundaries, respectively. A parser reads the first character/byte and then processes that value to get an index into a lookup table that it uses to find how many remaining characters/bytes to extract from the Stream. This makes the Primitive self-framing.

Count or Group Framing Codes

As mentioned above, one of the primary advantages of composable encoding is that special Framing Codes can be specified to support groups of Primitives. Grouping enables pipelining. Other suitable terms for these special Framing Codes are Group Codes or Count Codes for short. These are suitable terms because these Framing Codes can be used to count characters or bytes in a group of Primitives when parsing and offloading a Stream of CESR Primitives. A group of Primitives may be recursively composed into a group of groups.

A Count Code is its own composable Primitive, and its length, therefore, MUST be an integer multiple of four characters in the Text domain or, equivalently, an integer multiple of three bytes in the Binary domain. To clarify, a Count Code is a special Primitive that does not include a raw binary value, only its text code. Because a Count Code's raw binary element value is empty and its length MUST be an integer multiple of four characters (three bytes), its pad size MUST always be 0.

To elaborate, Count Codes MAY be used as separators to better organize a Stream of Primitives or to interleave non-native (non-CESR) serializations. Count Codes enable grouping any combination of Primitives, groups of Primitives, or non-native serializations to optimize pipelining and offloading.

Interleaved non-CESR serializations

Many applications use JSON [RFC4627], CBOR [RFC8949], or MessagePack (MGPK) [3] to serialize flexible self-describing data structures based on field maps, also known as dictionaries or hash tables.

CESR Primitives may also appear as a delimited text Primitive inside a non-native field map serialization. The delimited text may be either the key or value of a (key, value) pair.

In addition, one extremely useful property of CESR is that at the top-level of a CESR stream, non-native message serializations, namely, JSON, CBOR, and MGPK may be interleaved with native message serializations . This is NOT true for non-native serializations nested inside CESR groups framed by count codes, i.e., that are not at the top-level of a CESR stream.

When nesting inside CESR groups, a non-native CESR serializations MUST be encoded as a CESR primitive and then enclosed in a special count code for non-native messages.

Cold start Stream parsing problem

After a cold start, a Stream processor looks for framing information to know how to parse groups of elements in the Stream. If that framing information is ambiguous, then the parser may become confused and require yet another cold start. While processing a given Stream, a parser may become confused, especially if a portion of the Stream is malformed in some way. This usually requires flushing the Stream and forcing a cold start to

resynchronize the parser to subsequent Stream elements. A re-synchronization mechanism that does not require flushing the in-transit buffers but merely skipping to the next well-defined Stream element boundary in order to execute a cold start is a better option. Good cold start re-synchronization is essential to robust performant Stream processing.

For example, in Transmission Control Protocol (TCP), a cold start usually means closing and then reopening the TCP connection. This flushes the TCP buffers and sends a signal to the other end of the Stream that may be interpreted as a restart or cold start. In the User Datagram Protocol (UDP), each packet is framed individually, but a Stream may be segmented into multiple packets. So, a cold start may require an explicit ack or nack to force a restart.

Special CESR Count Codes support re-synchronization at each boundary between interleaved CESR and other serializations (like JSON, CBOR, or MGPK).

Performant resynchronization with unique start bits

A CESR Stream parser MUST support three specific interleaved serializations, namely, JSON, CBOR, and MGPK. To make the parser more performant and robust, fine-grained serialization boundary detection may be highly beneficial for interleaving these serializations in a CESR stream. One way to provide this is by selecting the Count Code start bits such that there is always a unique (mutually distinct) set of start bits at each interleaved boundary between CESR, JSON, CBOR, and MGPK.

Furthermore, it may also be highly beneficial to support in-stride switching between interleaved CESR text-domain Streams and CESR Binary domain Streams. In other words, the start bits for Count Codes in both the ‘T’ domain and the ‘B’ domain should be unique. This would provide the analogous equivalent of a UTF Byte Order Mark (BOM) [4]. Recall that a BOM enables a parser of UTF-encoded documents to determine if the UTF codes are big-endian or little-endian [4]. In the CESR case, an analogous feature would enable a Stream parser to know if a Count Code, along with its associated counted group of Primitives, is expressed in the ‘T’ or ‘B’ domain. Together these impose the constraint that the boundary start bits for interleaved text CESR, binary CESR, JSON, CBOR, and MGPK MUST be mutually distinct.

Only the first three bits of the codes for map objects in JSON, CBOR, and MGPK are fixed and not dependent on mapping size.

- In JSON, a serialized mapping object always starts with `{`. This is encoded as `0x7b`. the first three bits are `0b011`.
- In CBOR, the first three bits of the major type of its serialized mapping object are `0b101` corresponding to the bits denoting map “Major Type 5” from the CBOR spec.
- In MGPK, there are three different mapping object codes. The FixMap code starts with `0b100`. Both the Map16 and Map32 codes start with `0b110`.

Therefore, the JSON, CBOR, and MGPK encodings consume four starting Tritets (3 bits) that are in numeric order `0b011`, `0b100`, `0b101`, and `0b110`. This leaves four unused Tritets, namely, `0b000`, `0b001`, `0b010`, and `0b111`. These latter are potential candidates for the CESR Count Code start bits. In Base64, there are two codes that satisfy the constraints. The first is the dash character, `-`, encoded as `0x2d`. Its first three bits are `0b001`. The second is the underscore character, `_`, encoded as `0x5f`. Its first three bits are `0b010`. Both of these are distinct from the starting Tritets of any of the JSON, CBOR, and MGPK encodings above. Moreover, the starting Tritet of the corresponding binary encodings of `-` and `_` is `0b111`, which is also distinct from all the others. To elaborate, Base64 uses `-` in position 62 or `0x3E` (hex) and uses `_` in position 63 or `0x3F` (hex), both of which have starting Tritet of `0b111`

Consequently, two different Base64 characters, `-` and `_`, can be used for the first character of any Count Code in the 'T' domain. This also means there can be two different classes of Count Codes. Using Count Codes in this way also provides a BOM-like capability that enables a parser to determine if the Count Code itself is expressed in either the 'T' or 'B' domain. To clarify, if a Stream group starts with the Tritet `0b111`, then the Stream frame is 'B' domain CESR, and a Stream parser would thereby know how to convert the first sextet of the Stream group to determine which of the two Count Codes is being used, `0x3E` or `0x3F`. If, on the other hand, the Count Code starts with either of the Tritets `0b001` or `0b010`, then the Count Code is expressed in the 'T' domain, and a Stream parser likewise would thereby know how to convert the first character (octet) of the Count Code to determine which Count Code is being used for that group. Otherwise, if a Stream starts with `0b011`, then it is JSON. If it starts with `0b101`, then it is CBOR. If it starts with either `0b100` or `0b110`, then it is MGPK.

Finally, several useful applications of 'T' domain encoding of CESR streams for archiving or logging may benefit from annotations. The white space ASCII characters for line feed, carriage return, and tab all have starting tritets of `0b000`. Thus a parser would know to de-annotate such a stream before re-parsing.

The starting tritet of any cold start (restart) MUST begin with one of eight cases. These tritet start bit REQUIREMENTS are summarized in the following table:

Top-level Stream Starting Tritets

Starting Tritet	Serialization	Character
0b000	Annotated ‘T’ domain	
0b001	CESR ‘T’ domain Count Code	-
0b010	CESR ‘T’ domain Op Code	_
0b011	JSON	{
0b100	MGPK (FixMap)	
0b101	CBOR (Map “Major Type 5”)	
0b110	MGPK (Map16, Map32)	
0b111	CESR ‘B’ domain Count Code or Op Code	

Stream parsing rules

Given this set of Tritets (3 bits), a well-formed Stream start and restart requirement can be expressed.

Each Stream MUST start (restart) with one of eight cases:

1. A Count Code in CESR ‘T’ domain
2. A Count Code in CESR ‘B’ domain.
3. An Op code in the CESR ‘T’ domain
4. An Op code in the CESR ‘B’’ domain
5. A JSON encoded mapping.
6. A CBOR encoded mapping.
7. A MGPK encoded mapping.
8. Annotated Text domain CESR.

A parser merely needs to examine the first Tritet (3 bits) of the first byte of the Stream start to determine which one of the eight it is. When the first Tritet is a Count Code, then the remainder of the Count Code itself will include the additional information needed to parse the attached group. When the first Tritet indicates, its JSON, CBOR, or MGPK, the mapping’s first field MUST be a Version String that provides the additional information needed to parse the associated encoded field map serialization fully. See the Version

String annex for the detailed syntax of the information that may be extracted with a regular expression search, given that the first Tritet indicates the following bytes in the stream belong to a JSON, CBOR, or MGPK field map serialization.

The Stream MUST resume with a frame starting byte that begins with one of the 8 Tritets, either another Count Code expressed in the ‘T’ or ‘B’ domain or a new JSON, CBOR, or MGPK encoded mapping or a new annotated encoding.

This provides an extremely compact and elegant Stream parsing formula that generalizes support not only for CESR Composability but also for interleaved CESR with three of the most popular field map serializations.

Compact fixed-size codes

Typically, modern cryptographic suites support limited sets of raw binary Primitives with fixed (not variable) sizes. The design aesthetic is based on the understanding that there is minimally sufficient cryptographic strength and more cryptographic strength just wastes computation and bandwidth. Cryptographic strength is measured in bits of entropy, which also corresponds to the number of trials that have to be attempted to succeed in a brute-force attack. The accepted minimum for cryptographic strength is 128 bits of entropy or equivalently `2**128` (2 raised to the 128th power) brute force trials. The size in bytes of a given raw binary Primitive for a given modern cryptographic suite is usually directly related to this minimum strength of 128 bits (16 bytes).

For example, the raw binary Primitives from the well-known [6] ECC (Elliptic Curve Cryptography) library all satisfy this 128-bit strength goal. In particular, the digital signing public key raw binary Primitives for EdDSA are 256 bits (32 bytes) in length because well-known algorithms can reduce the number of trials to brute force invert an ECC public key to get the private key by the square root of the number of scalar multiplications which is also related to the size of both the private key and public key coordinates (discrete logarithm problem [5]). Therefore, 256-bit (32-byte) ECC keys are needed to achieve 128 bits of cryptographic strength. In general, the size of a given raw binary Primitive is typically some multiple of 128 bits of cryptographic strength. This is also true for the associated EdDSA raw binary signatures which are 512 bits (64 bytes) in length.

Similar scale factors exist for cryptographic digests. A standard default Blake3 digest is 256 bits (32 bytes) in length in order to get 128 bits of cryptographic strength. This is also true of SHA3-256. The sweet spots for modern cryptographic raw Primitive lengths are 32 bytes for many digests as well as EdDSA public and private keys as well as ECDSA private keys. Likewise, 64 bytes is the sweet spot for EdDSA and ECDSA-secp256k1 signatures and 64-byte variants of the most popular digests. Therefore, optimized text code tables for these two sweet spots (32 and 64 bytes) would be highly advantageous.

A 32-byte raw binary value has a pad size of 1 character.

```
((3 - (32 mod 3)) mod 3) = 1
```

Therefore, the minimal text code size is 1 character for 32-byte raw binary cryptographic material and all other raw binary material values whose pad size is 1 character.

A 64-byte raw binary value has a pad size of 2 characters.

```
((3 - (64 mod 3)) mod 3) = 2
```

Therefore, the minimal text code size is 2 characters for 64-byte raw binary cryptographic material and all other raw binary material values whose pad size is 1 character. For example, a 16-byte raw binary value also has a pad size of 2 characters.

For all other cryptographic material values whose pad size is 0, such as the 33-byte ECDSA public keys then, the minimum size text code is 4 characters. So, the minimally sized text code tables are 1, 2, and 4 characters, respectively.

As mentioned above, CESR uses a multiple-code table design that enables both size-optimized text codes for the most popular Primitive types and extensible universal support for all other Primitive types.

Given that a given Cryptographic Primitive type has a known fixed raw binary size, then that Primitive type and size can be encoded efficiently with just the type information. The size is given by the type.

For example, an Ed25519 (EdDSA) raw public key is always 32 bytes, so knowing that the type is `Ed25519 public key` implies the size of 32 bytes and a pad size of 1 character that, therefore, may be encoded with a 1-character text code. Likewise, an Ed25519 (EdDSA) signature is always 64 bytes, so knowing that the type is `Ed25519 signature` implies the size of 64 bytes and a pad size of 2 characters that, therefore, may be encoded with a 2-character text code.

Code table selectors

To efficiently parse a Stream of Primitives with types from multiple text code tables, the first character in the text code MUST determine which code table to use, either a default code table or a code table selector character when not the default code table. Thus, the 1-character text code table MUST do double duty. It MUST provide selectors for the different text code tables and MUST also provide type codes for the most popular Primitives that have a pad size of 1 that appears as the default code table.

There are 64 Base64 characters (64 values). Only 12 tables are needed to support all the codes and code formats needed for the foreseeable future. Therefore, only 12 of those characters need to be dedicated as code table selectors, which leaves 52 charac-

ters that may be used for the 1-character type codes in the default table. This gives a total of 13 type code tables consisting of the dual purpose 1 character type or selector code table and 12 other tables.

As described above, the selector characters for the Count Code tables that best support interleaved JSON, CBOR, and MGPK are `-` and `_`. The numerals `0` through `9` are used each to serve as a selector for the other tables. That leaves the letters `A` to `Z` and `a` to `z` as single character selectors. This provides 52 unique type codes for fixed-length Primitive types with raw binary values that have a pad size of 1.

To clarify, the first character of any Primitive MUST be either a selector or a 1-character code type. The characters `0` through `9`, `-`, and `_` are selectors that select a given code table and indicate the number of remaining characters in the text code.

In summary, a compliant CESR implementation MAY use up to 13 standard (non-special context) code tables. The Count Code table MUST use the '-' character as its selector. The Op Code table MUST use the '_' character as its selector. The single code character table MUST use the characters [A-Z,a-z]. The other 10 code tables MUST each use one the characters [0-9] as their selector.

Table types

The tables in CESR consist of:

- Two major types for raw CESR primitives:
- fixed-length raw size primitives
- variable-length raw size primitives
- Count code tables for grouping primitives
- Protocol genus and version tables for use in differentiating between genus and versions of protocols encoded in CESR
- Opcode tables that are not yet specified but may be used for advanced decentralized applications in the future
- Some special context specific code tables (and space for future tables) for use in various contexts, like easy indexing into arrays of primitives

The sections below explain these table types, the code selectors that differentiate these types, and how to use these tables to decode a given CESR stream.

Tables of Codes for Fixed-length Raw Sizes

Small fixed-length raw-size tables

There are two special tables that are dedicated to the most popular fixed-size raw binary Cryptographic Primitive types. These are the most compact, so they optimize bandwidth but only provide a small number of total types. In both of these, the code-specific size equals the number of pad characters, i.e., the pad size.

One-character fixed-length raw-size table

The one-character type code table does not have a selector character per se but uses as type codes the non-selector characters `A` - `Z` and `a` - `z`. This provides 52 unique type codes for fixed-size raw binary values with a pad size of 1.

Two-character fixed-length raw size table

The two-character type code table uses the selector `0` as its first character. The second character is the type code. This provides 64 unique type codes for fixed-size raw binary values that have a pad size of 2.

Large fixed-length raw-size tables

The three tables in this group are for large fixed raw-size Primitives. These three tables use 0, 1, or 2 lead bytes as appropriate for a pad size of 0, 1, or 2 for a given fixed raw binary value. The text code size for all three tables is 4 characters. The selector character not only encodes the table but also implicitly encodes the number of lead bytes. The 3 remaining characters are each type code in each table, providing 262,144 unique types. This should provide enough type codes to accommodate all fixed raw-size Primitive types for the foreseeable future.

Large fixed-length raw-size table with 0 Lead Bytes

This table uses `1` as its first character or selector. The remaining 3 characters provide the type of each code. Only fixed-size raw binaries with a pad size of 0 are encoded with this table. The 3-character type code provides a total of 262,144 unique type code values (`262144 = 64**3`) for fixed-size raw binary Primitives with a pad size of 0.

Large fixed-length raw-size table with 1 lead byte

This table uses `2` as its first character or selector. The remaining 3 characters provide the type of each code. Only fixed-size raw binaries with a pad size of 1 are encoded with this table. The 3-character type code provides a total of 262,144 unique type code values (`262144 = 64**3`). Together with the 52 values from the 1-character code table above, there are 262,196 type codes for fixed-size raw binary Primitives with a pad size of 1.

Large fixed-length raw-size table with 2 lead bytes

This table uses `3` as its first character or selector. The remaining 3 characters provide the type of each code. Only fixed-size raw binaries with a pad size of 2 are encoded with this table. The 3-character type code provides a total of 262,144 unique type code values (`262144 = 64**3`). Together with the 64 values from the 2-character code table above (selector `0`), there are 262,208 type codes for fixed-size raw binary Primitives with a pad size of 2.

Tables for Codes with Variable-length Raw-sizes

Small variable-length raw-size tables

Although many Primitives have fixed raw binary sizes, especially those for modern cryptographic suites such as keys, signatures, and digests, there are other Primitives that benefit from variable sizings such as either encrypted material or legacy cryptographic material types found in the GNU Privacy Guard (GPG) or Open Secure Sockets Layer (OpenSSL) libraries (like the Rivest-Shamir-Adleman (RSA) algorithm. Furthermore, CESR is meant to support not merely cryptographic material types but other basic types, such as generic text strings and numbers. These basic non-cryptographic types may also benefit from variable-size codes.

The three tables in this group are for small variable raw-size Primitives. These three tables use 0, 1, or 2 lead bytes as appropriate given the pad size of 0, 1, or 2 for a given variable size raw binary value. The text code size for all three tables is 4 characters. The first character is the selector, the second character is the type, and the last two characters provide the size of the value as a Base64 encoded integer.

The number of unique type codes in each table is, therefore, 64. A given type code is repeated in each table for the same type so that all that differs between codes of the same type in each table is the number of lead bytes needed to align a given variable length on a 24-bit boundary. To clarify, what is different in each table is the number of lead bytes needed to achieve 24-bit alignment for a given variable length. Thus, the selector not only encodes for which type table but also implicitly encodes the number of lead bytes. The variable size is measured in Quadlets of 4 characters each in the ‘T’ domain and equivalently in triplets of 3 bytes each in the ‘B’ domain. Thus, computing the number of characters when parsing or off-loading in the ‘T’ domain means multiplying the variable size by 4. Computing the number of bytes when parsing or off-loading in the ‘B’ domain means multiplying the variable size by 3. The two Base64 size characters provide value lengths in Quadlets/triplets from 0 to 4095 (`64**2 -1`). This corresponds to value lengths of up to 16,380 characters (`4095 * 4`) or 12,285 bytes (`4095 * 3`).

Small variable-length raw-size table with 0 lead bytes

This table uses `4` as its first character or selector. The second character provides the type. The final two characters provide the size of the value in Quadlets/triplets as a Base64 encoded integer. Only raw binaries with a pad size of 0 are encoded with this ta-

ble. The 1-character type code provides a total of 64 unique type code values. The maximum length of the value provided by the 2 size characters is 4095 Quadlets of characters in the ‘T’ domain and triplets of bytes in the ‘B’ domain. All are raw binary Primitives with a pad size of 0 that each includes 0 lead bytes.

Small variable-length raw-size table with 1 lead byte

This table uses `5` as its first character or selector. The second character provides the type. The final two characters provide the size of the value in quadlets/triplets as a Base64 encoded integer. Only raw binaries with a pad size of 1 are encoded with this table. The 1-character type code provides a total of 64 unique type code values. The maximum length of the value provided by the 2 size characters is 4095 quadlets of characters in the ‘T’ domain and triplets of bytes in the ‘B’ domain. All are raw binary Primitives with a pad size of 1 that each includes 1 lead byte.

Small variable-length raw-size table with 2 lead bytes

This table uses `6` as its first character or selector. The second character provides the type. The final two characters provide the size of the value in quadlets/triplets as a Base64 encoded integer. Only raw binaries with a pad size of 0 are encoded with this table. The 1-character type code provides a total of 64 unique type code values. The maximum length of the value provided by the 2 size characters is 4095 quadlets of characters in the ‘T’ domain and triplets of bytes in the ‘B’ domain. All are raw binary Primitives with a pad size of 2 that each includes 2 lead bytes.

Large Variable-length Raw-size Tables

Many legacy cryptographic libraries such as OpenSSL and GPG support any variable-sized Primitive for keys, signatures, and digests such as RSA. Although this approach is often criticized for providing too much flexibility, many legacy applications depend on this degree of flexibility. Consequently, these large variable raw size tables provide a sufficiently expansive set of tables with enough types and sizes to accommodate all the legacy cryptographic libraries as well as all the variable-sized non-cryptographic raw Primitive types for the foreseeable future.

> ISSUE
>
> Post-quantum cryptographic operations

The three tables in this group are for large variable raw size Primitives. These three large variable raw size tables use 0, 1, or 2 lead bytes as appropriate for the associated pad size of 0, 1, or 2 for a given variable-sized raw binary value. The text code size for all three tables is 8 characters. As a special case, the first 62 entries in these tables represent that same crypto suite type as the 62 entries in the small variable raw size tables above. This allows one type to use a smaller 4-character text code when the raw size is

small enough. The first character is the selector, the next three characters provide the type, and the last four characters provide the size of the value as a Base64 encoded integer.

With 3 characters for each unique type code, each table provides 262,144 unique type codes. This should be enough type codes to accommodate all fixed raw size Primitive types for the foreseeable future. A given type code is repeated in each table for the same type. What is different for each table is the number of lead bytes needed to align a given length on a 24-bit boundary. The selector not only encodes the table but also implicitly encodes the number of lead bytes. The variable size is measured in quadlets of 4 characters each in the ‘T’ domain and equivalently in triplets of 3 bytes each in the B’ domain. Thus, computing the number of characters when parsing or off-loading in the ‘T’ domain means multiplying the variable size by 4. Likewise, computing the number of bytes when parsing or off-loading in the ‘B’ domain means multiplying the variable size by 3. The four Base64 size characters provide value lengths in Quadlets/triplets from 0 to 16,777,215 (`64**4 -1`). This corresponds to value lengths of up to 67,108,860 characters (`16777215 * 4`) or 50,331,645 bytes (`16777215 * 3`).

Large variable-length raw-size table with 0 lead bytes

This table uses `7` as its first character or selector. The next three characters provide the type. The final four characters provide the size of the value in Quadlets/triplets as a Base64 encoded integer. Only raw binaries with a pad size of 0 are encoded with this table. The 3-character type code provides a total of 262,144 unique type code values. The maximum length of the value provided by the 4 size characters is 16,777,215 Quadlets of characters in the ‘T’ domain and triplets of bytes in the ‘B’ domain. All are raw binary Primitives with pad size of 0 that each includes 0 lead bytes.

Large variable-length raw-size table with 1 lead byte

This table uses `8` as its first character or selector. The next three characters provide the type. The final four characters provide the size of the value in Quadlets/triplets as a Base64 encoded integer. Only raw binaries with a pad size of 1 are encoded with this table. The 3-character type code provides a total of 262,144 unique type code values. The maximum length of the value provided by the 4 size characters is 16,777,215 Quadlets of characters in the ‘T’ domain and triplets of bytes in the ‘B’ domain. All are raw binary Primitives with a pad size of 1 that each includes 1 lead byte.

Large variable-length raw-size table with 2 lead bytes

This table uses `9` as its first character or selector. The next three characters provide the type. The final four characters provide the size of the value in Quadlets/triplets as a Base64 encoded integer. Only raw binaries with a pad size of 2 are encoded with this table. The 3-character type code provides a total of 262,144 unique type code values. The

maximum length of the value provided by the 4 size characters is 16,777,215 Quadlets of characters in the ‘T’ domain and triplets of bytes in the ‘B’ domain. All are raw binary Primitives with a pad size of 2 that each includes 2 lead bytes.

Count Code tables

All Count Codes except the genus/version code table (see below) are pipelineable because they count the number of Quadlets/triplets in the count group. A Quadlet is four Base64 characters in the Text domain. A Triplet is three B2 bytes in the Binary domain. Because this corresponds to the 24-bit alignment constraint of Composability defined above, the count value is invariant between ‘T’ and ‘B’ Domains. Therefore, the count value MUST be invariant in either Domain and MUST be the number of Quadlets in the ‘T’ domain and the number of Triplets in the ‘B’ domain.

This invariance allows a stream parser to extract the number of characters/bytes in a group from the Stream without parsing the group’s contents; it is therefore pipeline-able. By making all Count Codes pipeline-able, the Stream parser can be optimized in a granular way, including granular core affinity.

There may be as many at 13 Count Code tables, but only three are specified in the current version. These three are:

- The small count, four-character table
- The large count, eight-character table
- The eight-character protocol genus and version table.

Each Count Code MUST be aligned on a 24-bit boundary. Count Codes MUST NOT have a value component but MUST have only type and size components. The size component MUST count the Quadlets/triplets in its following group. Moreover, because Primitives are already guaranteed to be composable, Count Codes do not need to account for pad size because the Count Code MUST be aligned on a 24-bit boundary. The Count Code type indicates the type of Primitive or group being counted but always counts the number of quadlets/triplets in the group not the number of primitives. Each element in content of a Count Code group MUST be aligned on a 24-bit boundary. Thus the only elements allowed in the contents of a Count Code group are other primitives or groups.

Count Code tables MAY use a nested set of selectors. The first selector MUST always be the `-` character as the initial selector. The following character MAY be either a selector for another Count Code table or MAY be the type for the small Count Code table. When the second character is numeral `0` - `9` or the letters `-` or `_`, then it MUST be a secondary Count Code table selector. When the second character is a letter in the range `A` - `Z` or `a` - `z`, then it MUST be a unique single-character Count Code. This results in a

total of 52 single-character Count Codes. If at some time in the future, no more than the initial three count code tables are needed, then the number of single-character Count Codes could be expanded to include the unused selector codes.

Small Count Code table

Codes in the small Count Code table MUST be each four characters long. The first character MUST be the selector `-`. The second character MUST be the Count Code type. the last two characters MUST be the count size as a Base64 encoded integer. The Count Code type MUST be a letter `A` - `Z` or `a` - `z`. The set of letters provides 52 unique Count Codes. A two-character size provides counts from 0 to 4095 (`64**2 - 1`).

If the second character is not a letter but is a numeral `0` - `9` or `-` or `_`, then it MUST be either a selector for a different Count Code table or an error.

Large Count Code table

Codes in the large Count Code table MUST be each 8 characters long. The first two characters MUST be the selectors `--`. The next character MUST be the Count Code type. The last five characters MUST be the count size as a Base64 encoded integer. With one character for type, there are 64 unique large-count code types. A five-character size provides counts from 0 to 1,073,741,823 (`64**5 - 1`). These correspond to groups of size `1,073,741,823 * 4 = 4,294,967,292` characters or `1,073,741,823 * 3 = 3,221,225,469` bytes.

Protocol genus/version table

The protocol genus/version table is special because its codes modify the Count Code groups that MAY appear at the top level of the stream or the Count code groups that MAY appear inside three special (universal) enclosing Count Code groups. A protocol genus and version code itself MUST NOT provide a count of the following Quadlets or triplets but MUST modify the protocol genus and Version of all the following Count Codes that either appear at the top level until another protocol and genus Count Code is provided or are inside a special enclosing Count Code group. Of the universal count codes, there are three general-purpose special enclosing Count Codes that allow an embedded genus/version table code. These are defined below. Universal count codes MUST be universal across all genera of Count Code tables.

The purpose of the protocol genus/version table is twofold. First, it allows CESR to be used for different protocols and protocol stacks, where each protocol may have its own dedicated set of code tables. The only table that all protocols MUST share (i.e., has identical values) is the protocol genus and version table (protocol table for short). All small and large count code tables must share all the universal count codes (these are defined below). All other entries in the small and large count code tables and all other tables MAY

vary by protocol. Secondly, for a given protocol genus, a protocol genus and version code MUST provide the Version of that given protocol's table set. This allows versioning of the CESR code tables for a given protocol.

Protocol genus/version codes

The format for a protocol genus/version code MUST be as follows: `-_GGGVVV` where `GGG` represents the protocol genus and `VVV` represents the Version of that protocol genus. The genus uses three Base64 characters for a possible total of 262,144 different protocol genera. The next three characters, `VVV`, provide, in Base64 notation, the major and minor version numbers of the Version of the protocol genus. The first `V` character provides the major version number, and the final two `VV` characters provide the minor version number. For example, `CAA` indicates major version 2 and minor version 00 or in dotted-decimal notation, i.e., `2.00`. Likewise, `CAQ` indicates major version 2 and minor version decimal 16 or in dotted-decimal notation `2.16`. The version part supports up to 64 major versions with 4096 minor versions per major version.

Any addition of a new code to the code table is backward-breaking in at least one direction, so it is a feature change in at least one direction. New implementations with the new codes can accept streams from old implementations, but old ones will break if they receive the new ones.

A Major change occurs when a code's meaning changes. When a Major change occurs, the Major version number MUST be incremented. This means it breaks in both directions, i.e., sender and receiver.

A minor change occurs when a code is added to a table; this only breaks backward compatibility when a new sender sends to an old receiver, but a new sender will still correctly process a stream sent from an old receiver. Since code additions will be commonly compared to code changes, it is beneficial to have more room for minor vs. major versions. When a minor change occurs, the Minor version number MUST be incremented.

OpCode tables

Op Code table

The `_` selector MUST be reserved for the yet-to-be-defined opcode table or tables. Opcodes are designed to provide Stream processing instructions that are more general and flexible than simply concatenating Primitives or groups of Primitives. A yet-to-be-determined stack-based virtual machine could be executed using a set of opcodes that provide Primitive, groups of Primitives, or Stream processing instructions. This would enable highly customizable uses for CESR.

Summary of Selector code tables and encoding scheme design

Encoding scheme table

A given CESR protocol genus MUST use the table encoding schemes defined above and summarized in the following table:

The following table summarizes the ‘T’ domain coding schemes by selector code for the 15 code tables defined in the sections above:

Encoding Scheme Table

Table	Universal Selector	Selector	Type Chars	Value Size Chars	Code Size	Lead Bytes	Pad Size	Format
1-char fixed	`[A-Z,a-z]`		`1*`	0 or special	1	0	1	`$&&&`
2-char fixed	`0`		1	0 or special	2	0	2	`*$&&`
large fixed 0-char lead byte	`1`		3	0 or special	4	0	0	`*$$$&&&&`
large fixed 1-char lead byte	`2`		3	0 or special	4	1	1	`*$$$%&&&`
large fixed 2-char lead byte	`3`		3	0 or special	4	2	2	`*$$$%%&&`
small var 0-char lead byte	`4`		1	2	4	0	0	`*$##&&&&`

Table	Universal Selector	Selector	Type Chars	Value Size Chars	Code Size	Lead Bytes	Pad Size	Format
small var 1-char lead byte	`5`		1	2	4	1	1	`*$##%&&&`
small var 2-char lead byte	`6`		1	2	4	2	2	`*$##%%&&`
large var 0-char lead byte	`7`		3	4	8	0	0	`*$$$####&&&&`
large var 1-char lead byte	`8`		3	4	8	1	1	`*$$$####%&&&`
large var 2-char lead byte	`9`		3	4	8	2	2	`*$$$####%%&&`
small cnt code	`-`	`[A-Z,a-z]`	`1*`	0	4	0	0	`*$##`

Table	Universal Selector	Selector	Type Chars	Value Size Chars	Code Size	Lead Bytes	Pad Size	Format
large code cnt	`-`	`-`	1	0	8	0	0	`**$#####`
proto + genus	`-`	`_`	1	0	8	0	0	`**$$$###`
other cnt codes	`-`	`[0-9]`	TBD	TBD	TBD	TBD	TBD	`**`
op codes	`_`		TBD	TBD	TBD	TBD	TBD	`*`

Special fixed-size codes MAY convey values in the value size part of the code. This enables compact encoding of small special values like field tags, types, or versions. In this case, the Code Size MUST equal the size of the Selector, Type, and Value Size parts summed together. This means the converted raw part MAY be empty. In that case, a fixed-sized code but with a non-empty Value Size, the value of the Value Size part MAY have special meaning.

Encoding scheme symbols

The following table defines the meaning of the symbols used in the encoding scheme table `Format` column above:

Encoding Scheme Symbols Table

Symbol	Description
`*`	selector-code character also provides the type
`$`	type-code character from subset of Base64 `[A–Z, a–z,0–9,–,_]`
`%`	lead byte where pre-converted binary includes the number of lead bytes shown
`#`	Base64 digit as part of a base 64 integer. When part of Primitive determines the number of following Quadlets or triplets. When part of a Count Code determines the count of the following Primitives or groups of Primitives
`&`	Base64 value characters that represent the converted raw binary value. The actual number of characters is determined by the prepended text code. Shown is the minimum number of value characters.
`TBD`	to be determined, reserved for future use

Special context-specific code tables

The set of tables above provides the basic or master encoding schemes. These coding schemes constitute the basic or master set of code tables. This basic or master set, however, may be extended with context-specific code tables. The context in which a Primitive occurs may provide an additional implicit selector that is not part of the actual explicit text code. This allows context-specific coding schemes that would otherwise conflict with the basic or master encoding schemes and tables.

Indexed codes

Currently, only one context-specific coding scheme is defined. This is for indexed signatures. A given CESR Protocol Genus MAY define other context-specific coding schemes. The common use case for indexed signatures is thresholded multi-signature schemes. A

threshold-satisficing subset of signatures belonging to an ordered set or list of public keys may be provided as part of a Stream of Primitives. One way to compactly associate each signature with its public key is to include the index into the ordered set of public keys in the text code for that signature.

A popular signature raw binary size is 64 bytes, with a pad size of 2. This gives two code characters for a compact text code. The first character is the selector and type code. The second character is the Base64 encoded integer index. Using a similar dual selector type code character scheme as above, the selectors are the numbers `0-9` and `-` and `_`. Then there are 52 type codes given by the letters `A- Z` and `a-z`. The index has 64 values, which support up to 64 members in the public key list. A selector can select a large text code with more characters dedicated to larger indices. Some applications of CESR, such as KERI, require dual-indexed signatures (i.e., each signature has two indices) to support pre-rotation with partial or reserved participants in a rotation. With partial rotation, a given signature may contribute to the signing threshold for two different thresholds, each on two different lists of keys, where the associated key may appear at a different location in each list. In the tables below, for the codes that support dual indices, those indices are labeled as Index and Ondex, i.e., other-index.

For 64-byte signatures, the Ed25519 and ECDSA secp256k1 schemes have entries in the table. For dual-indexed codes, the next larger code size that aligns a 64-byte signature on a 24-bit boundary is 6 characters. The table provides entries for dual-indexed 64-byte signatures. The code includes one selector character, one type character, and two each of two-character indices.

A new signature scheme based on Ed448 with 114-byte signatures is also supported. These signatures have a pad size of zero, so they require a four-character text code. The first character is the selector `0`, the second character is the type with 64 values, and the last two characters each provide a one-character index as a Base64 encoded integer with 64 different values. A big Version code consumes eight characters with one character for the selector, one for the type, and three for each of the dual indices.

Indexed code table

The associated indexed schemes are provided in the following table:

Selector	Type Chars	Index Chars	Ondex Chars	Code Size	Lead Bytes	Pad Size	Format
`[A-Z,a-z]`	`1*`	1	0	2	0	2	`$#&&`
`0`	1	1	1	4	0	0	`0$##&&&`
`2`	1	2	2	6	0	2	`2$####&&&&`
`3`	1	3	3	6	0	0	`3$#####&&&&`

Encoding scheme format symbol table

The following table defines the meaning of the symbols used in the Indexed Code table:

Symbol	Description
`*`	selector-code character also provides the type
`$`	type-code character from subset of Base64 `[A-Z,a-z,0-9,-,_]`
`%`	lead byte where pre-converted binary includes the number of lead bytes shown
`#`	Base64 digit as part of a base 64 integer. When part of Primitive determines the number of following Quadlets or triplets. When part of a Count Code determines the count of following Primitives or groups of Primitives
`&`	Base64 value characters that represent the converted raw binary value. The actual number of characters is determined by the prepended text code. Shown is the minimum number of value characters.
`TBD`	to be determined, reserved for future

Parsing via table design

Text domain parsing can be simplified by using a parse size table. A Text domain parser uses the first character selector code to look up the hard-size (Stable) portion of the text code. The parser then extracts hard-size characters from the text Stream. These characters form an index for the parse size table, which includes a set of sizes for the remainder of the Primitive. Using these sizes for a given code allows a parser to extract and convert a given Primitive. In the Binary domain, the same text parse table may be used, but each size value represents a multiple of a sextet of bits instead of Base64 characters. Example entries from that table are provided below. Two of the rows may always be calculated given the other 4 rows, so the table need only have 4 entries in each row. Thus, all basic Primitives may be parsed with one parse size table.

Table 1

Selector code size table

selector	hs
`B`	1
`0`	2
`5`	2

Table 2

Parse size table

Below is a snippet from the parse size table for some example codes:

hard sized index	hs	ss	vs	fs	ls	ps
`B`	1	0	43*	44	0	1
`0B`	2	0	86*	88	0	2*
`5A`	2	2	#	#	1	1*

Table 3

Parse table symbols

Symbol	Description
`*`	entry's size may be calculated from other sizes
`#`	entry's size may be calculated from extracted code characters given by other sizes

Table 4

Parse part sizes

The following table includes both labels of parts shown in the columns in the parse size table as well as parts that may be derived from the parse table parts or from transformations,

Label	Description
‘hs’	hard size in chars (fixed) part of code size
‘ss’	soft size in chars, (Variable) part of code size
‘os’	other size in chars, when soft part includes two Variable values
‘ms’	main size in chars, (derived) when soft part provides two Variable values where ms = ss – os
‘cs’	code size in chars (derived value), where cs = hs + ss
‘vs’	value size in chars
‘fs’	full size in chars where fs = hs + ss + vs
‘ls’	lead size in bytes to pre-pad raw binary bytes
‘ps’	pad size in chars Base64 encoded
‘rs’	raw size in bytes (derived) of binary value where rs is derived from `R(T)`
‘bs’	binary size in bytes (derived) where bs = ls + rs

Annex A

Code table entry policy

The policy for placing entries into the tables, in general, is in order of first needed first-entered basis. In addition, the compact code tables prioritize entries that satisfy the requirement that the associated cryptographic operations maintain at least 128 bits of cryptographic strength. This precludes the entry of many weak cryptographic suites into the compact tables. CESR's compact code table includes only best-of-class cryptographic operations along with common non-Cryptographic Primitive types. At the time of this writing, there is the expectation that the National Institute of Standards and Technology (NIST) soon will approve standardized post-quantum resistant cryptographic operations. When that happens, codes for the most appropriate post-quantum operations will be added. For example, Falcon appears to be among the leading candidates with open-source code already available.

Table format

The tables below have the following format:

Each table has 5 columns. These are as follows:

1. The Base64 Stable (hard) text code itself.
2. A description of what is encoded or appended to the code.
3. The length in characters of the code.
4. The length in characters of the count portion of the code.
5. The length in characters of the fully qualified Primitive, including code and its appended material or the number of elements in the group. This is empty when variable length.

Universal Code tables

All code tables for every protocol genus/version MUST implement the following tables:

Universal Code table genus/version codes

Code	Description	Code Length	Count Length	Total Length
	Universal Genus Version Codes			
`-_AAA###`	KERI/ACDC stack code table at genus `AAA`	8	3*	8

> NOTE
>
> * This isn't a count of items on the stream like others in the count code tables below. Instead its the length of the characters wherein the total number of KERI/ACDC stack code genuses could exist (denoted by `###`).

Universal Code table genus/version codes that allow genus/version override

All genera MUST have the following codes in their Count Code table. Should the first Group Code embedded in each of these groups be a genus/version code, then the parser MUST switch code tables to the code table given by that genus/version code. All of the codes in the following table support this genus/version override. All other codes MUST NOT support this feature and are characterized as non-overrideable codes.

The presence of a genus/version count code that appears as the first element within the framed material of any non-overrideable count code (universal or not) MUST have no special meaning as an override to the stream parser. In other words, the parser MUST only treat the genus/version count code, especially as an override, when it appears as the first count code within the framed material of an overrideable universal count code. Otherwise, there MUST be no special override meaning to the parser. To elaborate, the parser's interpretation of a genus/version code's presence as the first element of the framed material of a non-overrideable count code depends on the framed material context in which it appears.

For example, suppose some application uses a list (a universal but non-overrideable count code) with a genus/version code as its first element. From the perspective of the stream parser, the genus/version count code's appearance as the list's first element has no special override semantics, i.e., its presence provides no special override meaning to the parser.

All genera MUST have the following codes in their Count Code table.

Code	Description	Code Length	Count Length	Total Length
	Count Codes			
	Universal Count Codes that allow genus/version override			
`-A##`	Generic pipeline group up to 4,095 quadlets/triplets	4	2	4
`--A#####`	Generic pipeline group up to 1,073,741,823 quadlets/triplets	8	5	8
`-B##`	Message + attachments group up to 4,095 quadlets/triplets	4	2	4
`--B#####`	Message + attachments group up to 1,073,741,823 quadlets/triplets	8	5	8
`-C##`	Attachments only group up to 4,095	4	2	4

Code	Description	Code Length	Count Length	Total Length
	quadlets/triplets			
`--C#####`	Attachments only group up to 1,073,741,823 quadlets/triplets	8	5	8

Universal Code table genus/version codes that do not allow genus/version override

All genera MUST have the following codes in their Count Code table.

Code	Description	Code Length	Count Length	Total Length
	Universal Count Codes that do not allow genus/version override			
`-D##`	Datagram Stream Segment up to 4,095 quadlets/triplets	4	2	4
`--D#####`	Datagram Stream Segment up to 1,073,741,823 quadlets/triplets	8	5	8
`-E##`	ESSR wrapper signable up to 4,095 quadlets/triplets	4	2	4
`--E#####`	ESSR wrapper signable up to 1,073,741,823 quadlets/triplets	8	5	8
`-F##`	CESR native message	4	2	4

Code	Description	Code Length	Count Length	Total Length
	top-level fixed field signable up to 4,095 quadlets/triplets			
`--F#####`	CESR native message top-level fixed field signable up to 1,073,741,823 quadlets/triplets	8	5	8
`-G##`	CESR native message top-level field map signable up to 4,095 quadlets/triplets	4	2	4
`--G#####`	CESR native message top-level field map signable up to 1,073,741,823 quadlets/triplets	8	5	8
`-H##`	Message group for enclosed non-native message to	4	2	4

Code	Description	Code Length	Count Length	Total Length
	4,095 quadlets/triplets			
`--H#####`	Message group for enclosed non-native message up to 1,073,741,823 quadlets/triplets	8	5	8
`-I##`	Generic field map mixed types up to 4,095 quadlets/triplets	4	2	4
`--I#####`	Generic field map mixed types up to 1,073,741,823 quadlets/triplets	8	5	8
`-J##`	Generic list mixed types up to 4,095 quadlets/triplets	4	2	4
`--J#####`	Generic list mixed types up to 1,073,741,823	8	5	8

Code	Description	Code Length	Count Length	Total Length
	quadlets/triplets			

KERI/ACDC Protocol Stack Tables

These tables are specific to the KERI/ACDC protocol genus. A compliant implementation of KERI/ACDC MUST support the following codes

KERI/ACDC protocol genus version table

Code	Description	Code Length	Count Length	Total Length
	Universal Genus Version Codes			
`-_AAABAA`	KERI/ACDC protocol stack code table at genus `AAA` and Version `1.00`	8		8
`-_AAACAA`	KERI/ACDC protocol stack code table at genus `AAA` and Version `2.00`	8		8

> NOTE
>
> Unlike the code in the Universal Code Selector Table above, these represent *specific* instantiations of the protocol genus codes so their count lengths are 0.

Master code table for genus/version `-_AAACAA` (KERI/ACDC protocol stack Version 2.00)

This master table includes the REQUIRED Primitive and Count Code types for the KERI/ACDC protocol stack. This table only provides the codes for the KERI/ACDC protocol stack code table genus `AAA` at Version 2.00 given by the genus/version code = `-_AAACAA` KERI/ACDC 2.00. It is anticipated that the code tables for the KERI/ACDC/TSP protocol stack will not change much in the future after 2.00. Hopefully, there will never be a Version 3.00 because 2.00 was designed properly.

This master table includes both the Primitive and Count Code types. The types are separated by headers.

A compliant KERI/ACDC genus MUST have the following codes in its Primitive and Count code tables.

Code	Description	Code Length	Count Length	Total Length
	Count Codes			
	Universal Genus Version Codes			
`-_AAABAA`	KERI/ACDC protocol stack code table at genus `AAA` and Version `1.00` *	8		8
`-_AAACAA`	KERI/ACDC protocol stack code table at genus `AAA` and Version `2.00` *	8		8
	Universal Count Codes that allow genus/version override			
`-A##`	Generic pipeline group up to 4,095 quadlets/triplets	4	2	4
`--A#####`	Generic pipeline group up to 1,073,741,823	8	5	8

Code	Description	Code Length	Count Length	Total Length
	quadlets/triplets			
`-B##`	Message + attachments group up to 4,095 quadlets/triplets	4	2	4
`--B#####`	Message + attachments group up to 1,073,741,823 quadlets/triplets	8	5	8
`-C##`	Attachments only group up to 4,095 quadlets/triplets	4	2	4
`--C#####`	Attachments only group up to 1,073,741,823 quadlets/triplets	8	5	8
	Universal Count Codes that do not allow genus/version override			
`-D##`	Datagram Stream Segment up	4	2	4

Code	Description	Code Length	Count Length	Total Length
	to 4,095 quadlets/triplets			
`--D#####`	Datagram Stream Segment up to 1,073,741,823 quadlets/triplets	8	5	8
`-E##`	ESSR wrapper signable up to 4,095 quadlets/triplets	4	2	4
`--E#####`	ESSR wrapper signable up to 1,073,741,823 quadlets/triplets	8	5	8
`-F##`	CESR native message top-level fixed field signable up to 4,095 quadlets/triplets	4	2	4
`--F#####`	CESR native message top-level	8	5	8

Code	Description	Code Length	Count Length	Total Length
	fixed field signable up to 1,073,741,823 quadlets/triplets			
`-G##`	CESR native message top-level field map signable up to 4,095 quadlets/triplets	4	2	4
`--G#####`	CESR native message top-level field map signable up to 1,073,741,823 quadlets/triplets	8	5	8
`-H##`	Message group for enclosed non-native message to 4,095 quadlets/triplets	4	2	4
`--H#####`	Message group for enclosed non-native message up to	8	5	8

Code	Description	Code Length	Count Length	Total Length
	1,073,741,823 quadlets/triplets			
`-I##`	Generic field map mixed types up to 4,095 quadlets/triplets	4	2	4
`--I#####`	Generic field map mixed type up to 1,073,741,823 quadlets/triplets	8	5	8
`-J##`	Generic list mixed types up to 4,095 quadlets/triplets	4	2	4
`--J#####`	Generic list mixed types up to 1,073,741,823 quadlets/triplets	8	5	8
	Genus Specific Count Codes			
`-K##`	Indexed controller signature group up to	4	2	4

Code	Description	Code Length	Count Length	Total Length
	4,095 quadlets/triplets			
`--K#####`	Indexed controller signature group up to 1,073,741,823 quadlets/triplets	8	5	8
`-L##`	Indexed witness signature group up to 4,095 quadlets/triplets	4	2	4
`--L#####`	Indexed witness signature group up to 1,073,741,823 quadlets/triplets	8	5	8
`-M##`	Nontransferable identifier receipt couples pre+sig up to 4,095 quadlets/triplets	4	2	4
`--M#####`	Nontransferable identifier	8	5	8

Code	Description	Code Length	Count Length	Total Length
	receipt couples pre+sig up to 1,073,741,823 quadlets/triplets			
`-N##`	Transferable identifier receipt quadruples pre+snu+dig+sig up to 4,095 quadlets/triplets	4	2	4
`--N#####`	Transferable identifier receipt quadruples pre+snu+dig+sig up to 1,073,741,823 quadlets/triplets	8	5	8
`-0##`	First seen replay couples fnu+dt up to 4,095 quadlets/triplets	4	2	4
`--0#####`	First seen replay couples fnu+dt up to	8	5	8

Code	Description	Code Length	Count Length	Total Length
	1,073,741,823 quadlets/triplets			
`-P##`	Pathed material group path+mixed-types up to 4,095 quadlets/triplets	4	2	4
`--P#####`	Pathed material group path+mixed-types up to 1,073,741,823 quadlets/triplets	8	5	8
`-Q##`	Digest seal singles `dig` up to 4,095 quadlets/triplets	4	2	4
`--Q#####`	Digest seal singles `dig` up to 1,073,741,823 quadlets/triplets	8	5	8
`-R##`	Merkle Tree Root seal singles	4	2	4

Code	Description	Code Length	Count Length	Total Length
	`rdig` up to 4,095 quadlets/triplets			
`--R#####`	Merkle Tree Root seal singles `rdig` up to 1,073,741,823 quadlets/triplets	8	5	8
`-S##`	Issuer/Delegator/Transaction event seal source couple snu+dig up to 4,095 quadlets/triplets	4	2	4
`-S#####`	Issuer/Delegator/Transaction event seal source couple snu+dig up to 1,073,741,823 quadlets/triplets	8	5	8
`-T##`	Anchoring event seal source triple pre+snu+dig up to 4,095	4	2	4

Code	Description	Code Length	Count Length	Total Length
	quadlets/triplets			
`--T#####`	Anchoring event seal source triple pre+snu+dig up to 1,073,741,823 quadlets/triplets	8	5	8
`-U##`	Last event seal source singles `aid+dig` up to 4,095 quadlets/triplets	4	2	4
`--U#####`	Last event seal source singles `aid+dig` up to 1,073,741,823 quadlets/triplets	8	5	8
`-V##`	Backer registrar identifier seal couples `brid+dig` up to 4,095 quadlets/triplets	4	2	4

Code	Description	Code Length	Count Length	Total Length
`--V#####`	Backer registrar identifier seal couples `brid+dig` up to 1,073,741,823 quadlets/triplets	8	5	8
`-W##`	Typed digest seal couples `type+dig` up to 4,095 quadlets/triplets	4	2	4
`--W#####`	Typed digest seal couples `type+dig` up to 1,073,741,823 quadlets/triplets	8	5	8
`-X##`	Transferable indexed sig group pre+snu+dig +idx-controller-sig-groups up to 4,095 quadlets/triplets	4	2	4
`--X#####`	Transferable indexed sig group	8	5	8

Code	Description	Code Length	Count Length	Total Length
	pre+snu+dig +idx-controller-sig-groups up to 1,073,741,823 quadlets/triplets			
`-Y##`	Transferable last indexed sig group pre+idx-controller-sig-groups up to 4,095 quadlets/triplets	4	2	4
`--Y#####`	Transferable last indexed sig group pre+idx-controller-sig-groups up to 1,073,741,823 quadlets/triplets	8	5	8
`-Z##`	ESSR (TSP) Payload `version+messagtype+...` up to 4,095 quadlets/triplets	4	2	4

Code	Description	Code Length	Count Length	Total Length
`--Z#####`	ESSR (TSP) Payload `version+messagtype+...` up to 1,073,741,823 quadlets/triplets	8	5	8
`-a##`	Blinded State quadruples dig+uuid+said+state up to 4,095 quadlets/triplets	4	2	4
`--a#####`	Big Blinded State quadruples dig+uuid+said+state up to 1,073,741,823 quadlets/triplets	8	5	8
`-b##`	Bound Blinded State Sextuples blid+uuid+said+state+bsnu+bsaid up to 4,095 quadlets/triplets	4	2	4
`--b#####`	Big Bound Blinded State Sextuples	8	5	8

Code	Description	Code Length	Count Length	Total Length
	blid+uuid+said+state+bsnu+bsaid up to 1,073,741,823 quadlets/triplets			
`-c##`	Typed and Blinded IANA media type quadruples blid+uuid+type+media up to 4,095 quadlets/triplets	4	2	4
`--c#####`	Big Typed and Blinded IANA media type quadruples blid+uuid+type+media up to 1,073,741,823 quadlets/triplets	8	5	8
	Operation Codes			
`_`	Reserved TBD			
	Primitive Matter Codes			

Code	Description	Code Length	Count Length	Total Length
	Basic One Character Codes			
A	Seed of Ed25519 private key	1		44
B	Ed25519 non-transferable prefix public verification key	1		44
C	X25519 public encryption key, may be converted from Ed25519 public key	1		44
D	Ed25519 public verification key	1		44
E	Blake3-256 Digest	1		44
F	Blake2b-256 Digest	1		44
G	Blake2s-256 Digest	1		44
H	SHA3-256 Digest	1		44
I	SHA2-256 Digest	1		44

Code	Description	Code Length	Count Length	Total Length
J	Seed of ECDSA secp256k1 private key	1		44
K	Seed of Ed448 private key	1		76
L	X448 public encryption key	1		76
M	Short number 2-byte b2	1		4
N	Big number 8-byte b2	1		12
O	X25519 private decryption key/seed may be converted from Ed25519 key/seed	1		44
P	X25519 124 char qb64 Cipher of 44 char qb64 Seed	1		124
Q	ECDSA secp256r1 256-bit random Seed for private key	1		44

Code	Description	Code Length	Count Length	Total Length
`R`	Tall 5-byte b2 number	1		8
`S`	Large 11-byte b2 number	1		16
`T`	Great 14-byte b2 number	1		20
`U`	Vast 17-byte b2 number	1		24
`V`	Label1 1 bytes for label lead size 1	1		4
`W`	Label2 2 bytes for label lead size 0	1		4
`X`	Tag3 3 B64 encoded chars for special values	1	3	4
`Y`	Tag7 7 B64 encoded chars for special values	1	7	8
`Z`	Tag11 11 B64 encoded chars for special values	1		12

Code	Description	Code Length	Count Length	Total Length
`a`	Blinding factor 256 bits, Cryptographic strength deterministically generated from random salt	1		44
Basic Two Character Codes				
`0A`	Random salt, seed, nonce, private key, or sequence number of length 128 bits	2		24
`0B`	Ed25519 signature	2		88
`0C`	ECDSA secp256k1 signature	2		88
`0D`	Blake3-512 Digest	2		88
`0E`	Blake2b-512 Digest	2		88
`0F`	SHA3-512 Digest	2		88
`0G`	SHA2-512 Digest	2		88

Code	Description	Code Length	Count Length	Total Length
`0H`	Long number 4-byte b2	2		8
`0I`	ECDSA secp256r1 signature	2		88
`0J`	Tag1 1 B64 encoded char + 1 prepad for special values	2	2	4
`0K`	Tag2 2 B64 encoded chars for for special values	2	2	4
`0L`	Tag5 5 B64 encoded chars + 1 prepad for for special values	2	6	8
`0M`	Tag6 6 B64 encoded chars for for special values	2	6	8
`0N`	Tag9 9 B64 encoded chars + 1 prepad for special values	2	10	12
`0O`	Tag10 10 B64 encoded	2	10	12

Code	Description	Code Length	Count Length	Total Length
	chars for special values			
`0P`	Gram Head Neck	2	22	32
`0Q`	Gram Head	2	22	28
`0R`	Gram Head AID Neck	2	22	76
`0S`	Gram Head AID	2	22	72
	Basic Four Character Codes			
`1AAA`	ECDSA secp256k1 non-transferable prefix public verification key	4		48
`1AAB`	ECDSA secp256k1 public verification or encryption key	4		48
`1AAC`	Ed448 non-transferable prefix public verification key	4		80
`1AAD`	Ed448 public verification key	4		80

Code	Description	Code Length	Count Length	Total Length
1AAE	Ed448 signature	4		156
1AAF	Tag4 4 B64 encoded chars for special values	4	4	8
1AAG	DateTime Base64 custom encoded 32 char ISO-8601 DateTime	4		36
1AAH	X25519 100 char b64 Cipher of 24 char qb64 Salt	4		100
1AAI	ECDSA secp256r1 verification key non-transferable, basic derivation	4		48
1AAJ	ECDSA secp256r1 verification or encryption key, basic derivation	4		48
1AAK	Null None or empty value	4		4

Code	Description	Code Length	Count Length	Total Length
`1AAL`	No falsey Boolean value	4		4
`1AAM`	Yes truthy Boolean value	4		4
`1AAN`	Tag8 8 B64 encoded chars for special values	4	8	12
	Variable Raw Size Codes			
`1AAO`	Escape code for escaping special map field values	4		4
`1AAP`	Empty value for nonce or string	4		4
`4A`	String Base64 Only Lead Size 0	4	2	
`5A`	String Base64 Only Lead Size 1	4	2	
`6A`	String Base64 Only Lead Size 2	4	2	
`7AAA`	String Big Base64 Only Lead Size 0	8	4	

Code	Description	Code Length	Count Length	Total Length
`8AAA`	String Big Base64 Only Lead Size 1	8	4	
`9AAA`	String Big Base64 Only Lead Size 2	8	4	
`4B`	Bytes Lead Size 0	4	2	
`5B`	Bytes Lead Size 1	4	2	
`6B`	Bytes Lead Size 2	4	2	
`7AAB`	Bytes Big Lead Size 0	8	4	
`8AAB`	Bytes Big Lead Size 1	8	4	
`9AAB`	Bytes Big Lead Size 2	8	4	
`4C`	X25519 sealed box cipher bytes of sniffable plaintext lead size 0	4	2	
`5C`	X25519 sealed box cipher bytes of sniffable plaintext lead size 1	4	2	
`6C`	X25519 sealed box cipher bytes	4	2	

Code	Description	Code Length	Count Length	Total Length
	of sniffable plaintext lead size 2			
7AAC	X25519 sealed box cipher bytes of sniffable plaintext big lead size 0	8	4	
8AAC	X25519 sealed box cipher bytes of sniffable plaintext big lead size 1	8	4	
9AAC	X25519 sealed box cipher bytes of sniffable plaintext big lead size 2	8	4	
4D	X25519 sealed box cipher bytes of QB64 plaintext lead size 0	4	2	
5D	X25519 sealed box cipher bytes of QB64 plaintext lead size 1	4	2	
6D	X25519 sealed box cipher bytes	4	2	

Code	Description	Code Length	Count Length	Total Length
	of QB64 plaintext lead size 2			
`7AAD`	X25519 sealed box cipher bytes of QB64 plaintext big lead size 0	8	4	
`8AAD`	X25519 sealed box cipher bytes of QB64 plaintext big lead size 1	8	4	
`9AAD`	X25519 sealed box cipher bytes of QB64 plaintext big lead size 2	8	4	
`4E`	X25519 sealed box cipher bytes of QB2 plaintext lead size 0	4	2	
`5E`	X25519 sealed box cipher bytes of QB2 plaintext lead size 1	4	2	
`6E`	X25519 sealed box cipher bytes	4	2	

Code	Description	Code Length	Count Length	Total Length
	of QB2 plaintext lead size 2			
`7AAE`	X25519 sealed box cipher bytes of QB2 plaintext big lead size 0	8	4	
`8AAE`	X25519 sealed box cipher bytes of QB2 plaintext big lead size 1	8	4	
`9AAE`	X25519 sealed box cipher bytes of QB2 plaintext big lead size 2	8	4	
`4F`	HPKE Base cipher bytes of QB2 plaintext lead size 0	4	2	
`5F`	HPKE Base cipher bytes of QB2 plaintext lead size 1	4	2	
`6F`	HPKE Base cipher bytes of QB2 plaintext lead size 2	4	2	

Code	Description	Code Length	Count Length	Total Length
7AAF	HPKE Base cipher bytes of QB2 plaintext big lead size 0	8	4	
8AAF	HPKE Base cipher bytes of QB2 plaintext big lead size 1	8	4	
9AAF	HPKE Base cipher bytes of QB2 plaintext big lead size 2	8	4	
4H	Decimal number string lead size 0	4	2	
5H	Decimal number string lead size 1	4	2	
6H	Decimal number string lead size 2	4	2	
7AAH	Decimal number string big lead size 0	8	4	
8AAH	Decimal number string big lead size 1	8	4	

Code	Description	Code Length	Count Length	Total Length
`9AAH`	Decimal number string big lead size 2	8	4	

> NOTE
>
> *Similar to the table above, these represent *specific* instantiations of the protocol genus codes so their count lengths are 0.
>
> --AAACAA may recieve a --AAABAA code on the stream but would have to parse with a table from the original reference implementation not included in this specification.

Indexed code table for genus/version `--AAACAA` (KERI/ACDC protocol stack version 2.00)

A compliant KERI/ACDC genus MUST have the following codes in its contextual indexed code table.

Code	Description	Code Length	Index Length	Ondex Length	Total Length
	Indexed Two Character Codes				
`A#`	Ed25519 indexed signature both same	2	1	0	88
`B#`	Ed25519 indexed signature current only	2	1	0	88
`C#`	ECDSA secp256k1 indexed sig both same	2	1	0	88
`D#`	ECDSA secp256k1 indexed sig current only	2	1	0	88
	Indexed Four Character Codes				
`0A##`	Ed448 indexed signature dual	4	1	1	156

Code	Description	Code Length	Index Length	Ondex Length	Total Length
`0B##`	Ed448 indexed signature current only	4	1	1	156
	Indexed Six Character Codes				
`2A####`	Ed25519 indexed sig big dual	6	2	2	92
`2B####`	Ed25519 indexed sig big current only	6	2	2	92
`2C####`	ECDSA secp256k1 indexed sig big dual	6	2	2	92
`2D####`	ECDSA secp256k1 idx sig big current only	6	2	2	92
	Indexed Eight Character Codes				
`3A######`	Ed448 indexed	8	3	3	160

Code	Description	Code Length	Index Length	Ondex Length	Total Length
	signature big dual				
`3B#####`	Ed448 indexed signature big current only	8	3	3	160

Legend:

Short name	Description
`pre`	Prefix
`sn`	Sequence number
`dig`	Digest
`sig`	Signature
`fn`	First seen number
`idx`	Index
`dt`	DateTime

Examples

The tables above include complex groups that maybe composed of other groups. For example, consider the counter transferable indexed signatures attachment group with code `-X##` where `##` is replaced by the two-character Base64 count of the number of complex groups. This is known as the TransIndexedSigGroups counter. Within the complex group are one or more attached groups where each TransIndexedSigGroup group consists of a triple pre+snu+dig followed by a ControllerIdxSigs group that in turn, consists of a Count Code `-K##` followed by one or more indexed signature Primitives each with indexed primitive code `A#`.

The following example details how a complex nested group may appear.

The example has only one group that includes one nested group. The example is annotated with comments, spaces, and line feeds for clarity.

```
-XBf  # Trans Indexed Sig Groups count Bf Quadlets in group
    EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB  # AID of signer
    0AAAAAAAAAAAAAAAAAAAAAAA  # sequence number of event being si
gned
    EPR7FWsN3tOM8PqfMap2FRfF4MFQ4v3ZXjBUcMVtvhmB  # SAID of event
being signed
    -KBC  # Controller Indexed Sigs count BC quadlets in group
        AADQ-rNV53XEXW1mI24X6uK3LlSMxqQxzM3HuWv_rbEkGP8kVjEYjzrBg
8o5hRCxXPnoO2zpHmh52OdUdog7xb0B  # signature 0
        ABCD_iSjAJvu9JsXHBAnCCTGCA-YSTKiRG-y6gUV42tzkL11OSEqRztXZ
Oq4yCBHcf4WTPT8fsMoaJGbW1a5JFkP  # signature 1
        ACBcPS0C_QwGdJUZTKXvC_qCs6069pqV8rdQymrJTdcmJAEYJDJXuHUc6
sjgdb0_VlPYIPtVZ9ypbRhkkuXJOykL  # signature 2
```

Version String field

Non-CESR serializations, namely, JSON, CBOR, and MGPK when interleaved in a CESR Stream MUST have a Version String as their first field with field label, `v` (lower case "v"). The Version String field value enables the Stream parser to use a regular expression parser to determine the type and length of the interleaved serialization. See the section on cold start stream processing section above for more detail on how a stream parser detects when to perform a regular expression search for a version string in a JSON, CBOR, or MGPK serialization interleaved in a CESR stream.

Version 2.XX string field format

The Version String, `v` field MUST be the first field in any top-level field map of any interleaved JSON, CBOR, or MGPK serialization. It provides a regular expression target for determining a serialized field map's serialization format and size (character count) of its enclosing field map. A Stream parser MUST be able to use the Version String to extract and deserialize (deterministically) any serialized Stream field maps. Each field map in a Stream MUST use one of the serialization types from the JSON, CBOR, or MGPK set. Each field map MAY have a different serialization type.

The format of the Version String is `PPPPMmmGggKKKKBBBB.`. It is 19 characters in length and is divided into five parts:

- Protocol: `PPPP` four character version string (for example, `KERI` or `ACDC`)
- Protocol Version: `Mmm` three character major/minor version of the protocol (described below)
- Genus Version: `Ggg` three character major/minor version of the CESR genus table (described below)
- Serialization kind: `KKKK` four character string of the types (`JSON`, `CBOR`, `MGPK`, `CESR`)

- Serialization length: `BBBB` integer encoded in Base64 equal to the number of characters (inclusive).
- version 2.XX terminator character `.`

The first four characters, `PPPP` indicate the protocol. Each genus of a given CESR code table set may support multiple protocols.

The next three characters, `Mmm`, provide in base 64 numerical notation the major and minor version numbers of the Version of the protocol specification. The first `V` character provides the major version number, and the final two `VV` characters provide the minor version number. For example, `CAA` indicates major version 2 and minor version decimal 0 or in dotted-decimal notation, i.e., `2.0`. Likewise, `CAQ` indicates major version 2 and minor version decimal 16 or in dotted-decimal notation `2.16`. The Version part supports up to 64 major versions with 4096 minor versions per major version.

The next three characters, `Ggg`, provide in base 64 numerical notation the major and minor version numbers of the Version of the CESR genus table used in the message. This assumes that for a given Protocol, the CESR genus is fixed and is determinable by the protocol, so only the genus version is needed. The first `G` character provides the major version number, and the final two `gg` characters provide the minor version number. For example, `CAA` indicates major version 2 and minor version 00 or in dotted-decimal notation, i.e., `2.0`. Likewise, `CAQ` indicates major version 2 and minor version decimal 16 or in dotted-decimal notation `1.16`. The Version part supports up to 64 major versions with 4096 minor versions per major version.

By base 64 numerical notation, we mean a number expressed as a base 64 number, not a string encoded in Base64. A single base 64 digit has 64 possible numeric values, `[0–63]` base 10, where `A` corresponds to 0 and `_` corresponds to 63. The numeric value comes from the index or offset of the character in the Base64 conversion table. To clarify, in base 64 numerical notation, each base 64 digit is a part of a number base 64, where the position in the number determines the base 64 exponent. For example, to convert a base 64 number to base 10, the digit in the zeroth position (right most) is multiplied by 64 raised to the exponent 0, i.e. is multiplied by 1, the digit in the first position is multiplied by 64 raised to the exponent 1, i.e. multiplied by 64, the second digit is multiplied by 64 raised to the exponent 2, i.e. multiplied by 4096. The position values are summed together to provide the equivalent base 10 numerical value. In the case of a version string, a 3-digit base 64 number is used as is without converting to base 10. One just needs to remember that each base 64 numerical digit has 64 numerical values `[0–63]` in base 10, which correspond to the index in the Base64 conversion table for each character.

The next four characters, `KKKK` indicate the serialization kind in uppercase. The four supported serialization kinds are `JSON`, `CBOR`, `MGPK`, and `CESR` for the JSON, CBOR, MessagePack, and CESR serialization standards, respectively [RFC4627] [RFC8949] [3] CESR. The last one, CESR is special. A CESR native serialization of a field map may

use either the `-G##` or `--G#####` count codes to indicate both that it is a field map and its size. Moreover, because count codes have unique start bits (see the section on Performant resynchronization) there is no need to embed a regular expression parsable version string field in a CESR native field map. Instead, a native CESR message's field map includes a protocol version field that indicates the protocol and version but not the size and serialization type. These are provided already by the count code. As a result, once deserialized into an in-memory data object representation of that field map, there is no normative indication that the in-memory object was deserialized from a CESR native field map (i.e. no version string field with serialization kind). This serialization kind indication would otherwise have to be provided externally. Instead, the in-memory object representation of the field map may inject a placeholder version string, `v` field, whose value is a version string but with the serialization kind set to `CESR`. This way, when re-serializing, there is a normative indicator to reserialize as a CESR native field map, not JSON, CBOR, or MGPK. This reserialization does not include an embedded version string field. It only appears in the in-memory object representation, not the serialization.

The next four characters, `BBBB`, provide in Base64 notation the total length of the serialization, inclusive of the Version String and any prefixed characters or bytes. This length is the total number of characters in the serialization of the field map. The maximum length of a given field map serialization is thereby constrained to be $64^4 = 2^{24} =$ 16,777,216 characters in length. This is deemed generous enough for the vast majority of anticipated applications. For serializations that may exceed this size, a secure hash chain of Messages may be employed where the value of a field in one Message is the cryptographic digest, SAID of the following Message. The total size of the chain of Messages may, therefore, be some multiple of 2^{24}.

The final character `.` is the Version String terminator. This enables later Versions of a protocol to change the total Version String size and thereby enable versioned changes to the composition of the fields in the Version String while preserving deterministic regular expression extractability of the Version String.

Although a given field map serialization kind may have characters or bytes such as field map delimiters or Framing Codes that appear before, i.e., prefix the Version String field in a serialization, the set of possible prefixes for each of the supported serialization kinds is sufficiently constrained by the allowed serialization protocols to guarantee that a regular expression can determine unambiguously the start of any ordered field map serialization that includes the Version String as the first field value. Given the length of the serialization provided by the Version String, a parser may then determine the end of the serialization to extract the full field map serialization from the Stream without first deserializing it. This enables performant Stream parsing and off-loading of Streams that include any or all of the supported serialization types.

Legacy Version 1.XX string field format

Compliant Version 2.XX implementations MUST support the old Version 1.XX Version String format to properly verify field maps created with 1.XX format events.

The format of the Version String for version 1.XX is `PPPPvvKKKKllllll_`. It is 17 characters in length and is divided into five parts:

- Protocol: `PPPP` four character version string (for example, `KERI` or `ACDC`)
- Version: `vv` two character major minor version (described below)
- Serialization kind: `KKKK` four character string of the types (`JSON`, `CBOR`, `MGPK`, `CESR`)
- Serialization length: `llllll` integer encoded in lowercase hexidecimal (Base 16) format
- legacy version terminator character `_`

The first four characters, `PPPP` indicate the protocol.

The next two characters, `vv` provide the major and minor version numbers of the Version of the protocol specification in lowercase hexadecimal notation. The first `v` provides the major version number, and the second `v` provides the minor version number. For example, `01` indicates major version 0 and minor version 1 or in dotted-decimal notation `0.1`. Likewise, `1c` indicates major version 1 and minor version decimal 12 or in dotted-decimal notation `1.12`.

The next four characters, `KKKK` indicate the serialization kind in uppercase. The four supported serialization kinds are `JSON`, `CBOR`, `MGPK`, and `CESR` for the JSON, CBOR, MessagePack, and CESR serialization standards, respectively [RFC4627] [RFC8949] [3] CESR.

The next six characters provide in lowercase hexadecimal notation the total length of the serialization, inclusive of the Version String and any prefixed characters or bytes. This length is the total number of characters in the serialization of the field map. The maximum length of a given field map serialization is thereby constrained to be $16^6 = 2^{24} =$ 16,777,216 characters in length. For example, when the length of serialization is 384 decimal characters/bytes, the length part of the Version String has the value `000180`. The final character `_` is the Version String terminator. This enables later Versions of the protocol to change the total Version String size and thereby enable versioned changes to the composition of the fields in the Version String while preserving deterministic regular expression extractability of the Version String.

Self-addressing identifier (SAID)

A SAID (Self-Addressing Identifier) is a special type of content-addressable identifier based on an encoded cryptographic digest that is self-referential. The SAID derivation protocol defined herein enables verification that a given SAID is uniquely cryptographically bound to a serialization that includes the SAID as a field in that serialization.

Embedding a SAID as a field in the associated serialization indicates a preferred content-addressable identifier for that serialization that facilitates greater interoperability, reduced ambiguity, and enhanced security when reasoning about the serialization. Moreover, given sufficient cryptographic strength, a cryptographic commitment such as a signature, digest, or another SAID, to a given SAID is essentially equivalent to a commitment to its associated serialization. Any change to the serialization invalidates its SAID thereby ensuring secure immutability evident reasoning with SAIDs about serializations or equivalently their SAID. Thus SAIDs better facilitate immutably referenced data serializations for applications such as Verifiable Credentials or Ricardian Contracts.

SAIDs MUST be encoded as a CESR [CESR] Primitive. As defined above, a CESR Primitive includes a pre-pended derivation code that encodes the cryptographic suite or algorithm used to generate the digest. A CESR Primitive's primary expression (alone or in combination) is textual using Base64 URL-safe characters. CESR Primitives may be round-tripped (alone or in combination) to a compact binary representation without loss. The CESR derivation code enables cryptographic digest algorithm agility in systems that use SAIDs as content addresses. Each serialization may use a different cryptographic digest algorithm as indicated by its derivation code. This provides interoperable future-proofing. CESR was developed for the KERI protocol.

The primary advantage of a content-addressable identifier is that it is cryptographically bound to the content (expressed as a serialization), thus providing a secure root-of-trust for reasoning about that content. Any sufficiently strong cryptographic commitment to a content-addressable identifier is functionally equivalent to a cryptographic commitment to the content itself.

A SAID is a special class of content-addressable identifier that is also self-referential. This requires a special derivation protocol that generates the SAID and embeds it in the serialized content. The reason for a special derivation protocol is that a naive cryptographic content-addressable identifier must not be self-referential, i.e., the identifier must not appear within the content that it is identifying. This is because the naive cryptographic derivation process of a content-addressable identifier is a cryptographic digest of the serialized content. Changing one bit of the serialization content will result in a different digest. Therefore, self-referential content-addressable identifiers require a special derivation protocol.

To elaborate, this approach of deriving self-referential identifiers from the contents they identify, is called `self-addressing`. It allows any validator to verify or re-derive the `self-referential, self-addressing identifier` given the contents it identifies. To clarify, a SAID is different from a standard content address or content-addressable identifier in that a standard content-addressable identifier is not included inside the contents it addresses. Moreover, a standard content-addressable identifier is computed on the fin-

ished immutable contents, and therefore is not self-referential. In addition, a SAID MUST include a pre-pended derivation code that specifies the cryptographic algorithm used to generate the digest. This provides Cryptographic agility .

An authenticatable data serialization is defined to be a serialization that is digitally signed with a non-repudiable asymmetric key-pair based signing scheme. A Verifier, given the public key, may verify the signature on the serialization and thereby securely attribute the serialization to the signer. Many use cases of authenticatable data serializations or statements include a self-referential identifier embedded in the authenticatable serialization. These serializations may also embed references to other self-referential identifiers to other serializations. The purpose of a self-referential identifier is to enable reasoning in software or otherwise about that serialization. Typically, these self-referential identifiers are not cryptographically bound to their encompassing serializations such as would be the case for a content-addressable identifier of that serialization. This poses a security problem because there now may be more than one identifier for the same content. The first is self-referential, included in the serialization, but not cryptographically bound to its encompassing serialization and the second is cryptographically bound but not self-referential, not included in the serialization.

When reasoning about a given content serialization, the existence of a non-cryptographically bound but self-referential identifier is a security vulnerability. Certainly, this identifier cannot be used by itself to securely reason about the content because it is not bound to the content. Anyone can place such an identifier inside some other serialization and claim that the other serialization is the correct serialization for that self-referential identifier. Unfortunately, a standard content-addressable identifier for a serialization which is bound to the serialization cannot be included in the serialization itself, i.e,. can be neither self-referential nor self-contained; it must be tracked independently. In contrast, a `self-addressing identifier` is included in the serialization to which it is cryptographically bound making it self-referential and self-contained. Reasoning about SAIDs is secure because a SAID will verify if and only if its encompassing serialization has not been mutated, which makes the content immutable. SAIDs used as references to serializations in other serializations enable tamper-evident reasoning about the referenced serializations. This enables a more compact representation of an authenticatable data serialization that includes other serializations by reference to their SAIDs instead of by embedded containment.

Generation and Verification Protocols

The SAID verification protocol MUST be implemented as follows:

- Make a copy of the embedded `CESR` [CESR] encoded SAID string included in the serialization.
- replace the SAID field value in the serialization with a dummy string of the same length. The dummy character is `#`, that is, ASCII 35 decimal (23 hex).

- Compute the digest of the serialization that includes the dummy value for the SAID field. Use the digest algorithm specified by the CESR [CESR] derivation code of the copied SAID.
- Encode the computed digest with CESR [CESR] to create the final derived and encoded SAID of the same total length as the dummy string and the copied embedded SAID.
- Compare the copied SAID with the recomputed SAID. If they are identical then the verification is successful; otherwise, unsuccessful.

Example Computation

The CESR [CESR] encoding of a Blake3-256 (32 byte) binary digest has 44 Base-64 URL-safe characters. The first character is `E` which represents Blake3-256. Therefore, a serialization of a fixed field data structure with a SAID generated by a Blake3-256 digest must reserve a field of length 44 characters. Suppose the initial value of the fixed field serialization is the 76-character string as follows:

```
field_0_01234567field_1_ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789field
_2_98765432
```

where:

field0 is the 16-character string “field_0_01234567” field1 is the 44-character placeholder string “field_1_ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789” field2 is the 16 character string “field_2_98765432”

The first step to generating the SAID for this serialization is to replace the placeholder contents of `field1` with a dummy string of `#` characters of length 44. This produces a dummied 76-character string as follows:

```
field_0_01234567############################################field
_2_98765432
```

The Blake3-256 digest is then computed on the above string and encoded in CESR format. This is the SAID of the 76-character string as follows:

```
ENI2bDYghiu1KYYkFrPofH8tJ5tNiNt8WrTIc4s_5IIH
```

Replacing the 44 dummy characters with the SAID of the same length produces the final SAIDified string as follows:

```
field_0_01234567ENI2bDYghiu1KYYkFrPofH8tJ5tNiNt8WrTIc4s_5IIHfield
_2_98765432
```

To verify the embedded SAID with respect to its encompassing serialization above, just reverse the generation steps. In other words, replace the SAID in the string with dummy characters of the same length, compute the Blake3 digest as SAID of this dummied version, and then compare the SAIDs.

Serialization Generation

Order-Preserving Data Structures

The crucial consideration in SAID generation is reproducibility. This requires the ordering and sizing of fields in the serialization to be fixed. Data structures in most computer languages have fixed fields. The example above is such an example.

A very useful type of serialization especially in some languages like Python or JavaScript is of self-describing data structures that are mappings of (key, value) or (label, value) pairs. These are often also called dictionaries or hash tables. The essential feature needed for reproducible serialization of such mappings is that mapping preserve the ordering of its fields on any round trip to/from a serialization. In other words, the mapping is ordered with respect to serialization. Another way to describe a predefined order preserving serialization is canonicalization or canonical ordering. This is often referred to as the mapping canonicalization problem.

The natural canonical ordering for such mappings is insertion order or sometimes called field creation order. Natural order allows the fields to appear in a preset order independent of the lexicographic ordering of the labels. This enables functional or logical ordering of the fields. Logical ordering also allows the absence or presence of a field to have meaning. Fields may have a priority given by their relative order of appearance. Fields may be grouped in logical sequence for better usability and labels may use words that best reflect their function independent of their relative lexicographic ordering. The most popular serialization format for mappings is JSON. Other popular serializations for mappings are CBOR and MessagePack.

In contrast, from a functional perspective, lexicographic ordering appears un-natural. In lexicographic ordering the fields are sorted by label prior to serialization. The problem with lexicographic ordering is that the relative order of appearance of the fields is determined by the labels themselves not some logical or functional purpose of the fields themselves. This often results in oddly-labeled fields that are so named merely to ensure that the lexicographic ordering matches a given logical ordering.

Originally mappings in most if not all computer languages were not insertion order preserving. The reason is that most mappings used hash tables under the hood. Early hash tables were highly efficient but by nature did not include any mechanism for preserving field creation or field insertion order for serialization. Fortunately, this is no longer true in general. Most if not all computer languages that support dictionaries or mappings as first-class data structures now support variations that are insertion order preserving.

For example, since Version 3.6 the default `dict` object in Python is insertion order preserving. Before that, Python 3.1 introduced the `OrderedDict` class which is insertion order preserving, and before that, custom classes existed in the wild for order preserving variants of a Python `dict`. Since Version 1.9 the Ruby version of a `dict`, the `Hash` class, is insertion order preserving. Javascript is a relative latecomer but since ECMAScript `ES6` the insertion ordering of JavaScript objects was preserved in `Reflect.ownPropertyKeys()`. Using custom `replacer` and `reviver` functions in `.stringify` and `.parse` allows one to serialize and de-serialize JavaScript objects in insertion order. Moreover, since ES11 the native `.stringify` uses insertion order all text string labeled fields in Javascript objects. It is an uncommon use case to have non-text string labels in a mapping serialization. A list is usually a better structure in those cases. Nonetheless, since ES6 the new Javascript `Map` object preserves insertion order for all fields for all label types. Custom `replacer` and `reviver` functions for `.stringify` and `.parse` allows one to serialize and de-serialize Map objects to/from JSON in natural order preserving fashion. Consequently, there is no need for any canonical serialization but natural insertion order preserving because one can always use lexicographic ordering to create the insertion order.

Example Python dict to JSON Serialization with SAID

Suppose the initial value of a Python `dict` is as follows:

```
{
    "said": "",
    "first": "Sue",
    "last": "Smith",
    "role": "Founder"
}
```

As before, the SAID will be a 44-character CESR encoded Blake3-256 digest. The serialization will be *JSON*. The `said` field value in the `dict` is to be populated with the resulting SAID. First the value of the `said` field is replaced with a 44-character dummy string as follows:

```
{
    "said": "############################################",
    "first": "Sue",
    "last": "Smith",
    "role": "Founder"
}
```

The `dict` is then serialized into JSON with no extra whitespace. The serialization is the following string:

```
{"said":"############################################","first":"S
ue","last":"Smith","role":"Founder"}
```

The Python code snippet for creating the JSON serialization is as follows:

```
import json

dummy = "#" * 44
dd = dict(said=dummy, first="Sue", last="Smith", role="Founder")
raw = json.dumps(dd, separators=(",", ":"), ensure_ascii=False)
assert raw == '{"said":"############################################
###","first":"Sue","last":"Smith","role":"Founder"}'
```

The Blake3-256 digest is then computed on that serialization above and encoded in CESR to provide the SAID as follows:

```
EJymtAC4piy_HkHWRs4JSRv0sb53MZJr8BQ4SMixXIVJ
```

The value of the `said` field is now replaced with the computed and encoded SAID to produce the final serialization with embedded SAID as follows:

```
{"said":"EJymtAC4piy_HkHWRs4JSRv0sb53MZJr8BQ4SMixXIVJ","first":"S
ue","last":"Smith","role":"Founder"}
```

The final serialization may be converted to a python `dict` by deserializing the JSON to produce:

```
{
    "said": "EJymtAC4piy_HkHWRs4JSRv0sb53MZJr8BQ4SMixXIVJ",
    "first": "Sue",
    "last": "Smith",
    "role": "Founder"
}
```

The generation steps may be reversed to verify the embedded SAID. The SAID generation and verification protocol for mappings assumes that the fields in a mapping serialization such as JSON are ordered in Stable, round-trippable, reproducible order, i.e., canonical. The natural canonical ordering is called field insertion order.

Example Schema Immutability using JSON Schema with SAIDs

SAIDs make JSON Schema fully self-contained with self-referential, unambiguously cryptographically bound, and verifiable content-addressable identifiers. The SAID derivation protocol defined above is applied to generate the `$id` field.

Given a Python dict of the schema: First, replace the value of the `$id` field with a string filled with dummy characters of the same length as the eventual derived value for `$id`.

```
{
    "$id": "############################################",
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "properties":
    {
        "full_name":
        {
           "type": "string"
        }
    }
}
```

Second, serialize the Python dict into JSON with no white space as follows:

```
import json

raw = json.dumps(dummied_schema-dict, separators=(",", ":"), ensu
re_ascii=False)
```

The serialization as a Python byte string is as follows:

```
(b'{"$id":"############################################","$schem
a":"http://json'
b'-schema.org/draft-07/schema#","type":"object","properties":{"fu
ll_name":{"ty'
b'pe":"string"}}}')
```

Third, make a digest of the serialized schema contents `raw` above and encode in CESR format. In this case this produces the schema SAID as follows:

```
EGU_SHY-8ywNBJOqPKHr4sXV9tOtOwpYzYOM63_zUCDW
```

Third, replace the dummy identifier value with the derived identifier value in the schema contents.

```
{
    "$id": "EGU_SHY-8ywNBJOqPKHr4sXV9tOtOwpYzYOM63_zUCDW",
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "properties": {
        "full_name": {
            "type": "string"
        }
    }
}
```

When a SAID is used for some field map data structure the enclosing data-structure is called self-addressing data (SAD).

Discussion

As long as any verifier recognizes the derivation code of a SAID, the SAID is a cryptographically secure commitment to the contents in which it is embedded; it is a cryptographically verifiable, self-referential, content-addressable identifier. Because a SAID is both self-referential and cryptographically bound to the contents it identifies, anyone can validate this binding if they follow the *derivation protocol* outlined above.

To elaborate, this approach of deriving self-referential identifiers from the contents they identify, is called `self-addressing`. It allows any validator to verify or re-derive the self-referential, self-addressing identifier given the contents it identifies. To clarify, a SAID is different from a standard content address or content-addressable identifier in that a standard content-addressable identifier is not included inside the contents it addresses. Moreover, a standard content-addressable identifier is computed on the finished immutable contents, and therefore is not self-referential.

Self-addressing Data (SAD) Path Signatures

SAD Path signatures are an extension to CESR that provide transposable cryptographic signature Attachments on self-addressing data (SAD). Any SAD, such as an Authentic Chained Data Container (ACDC) Verifiable Credential ACDC for example, may be signed with a SAD Path Signature and streamed along with any other CESR content. In addition, a signed SAD can be embedded inside another SAD and the SAD Path signature attachment can be transposed across envelope boundaries and streamed without losing any cryptographic integrity.

CESR is a dual text-binary encoding format that has the unique property of text-binary concatenation composability. The CESR specification not only provides the definition of the streaming format but also the attachment codes needed for differentiating the types of cryptographic material (such as signatures) used as attachments on all event types for the KERI [1]. While all KERI event Messages are SADs, there is a broad class of SADs that are not KERI events but that require signature attachments. ACDC Verifiable

Credentials fit into this class of SADs. With more complex data structures represented as SADs, such as Verifiable Credentials, there is a need to provide signature attachments on nested subsets of SADs. Similar to indices in indexed controller signatures in KERI that specify the location of the public key that they represent, nested SAD signatures need a path mechanism to specify the exact location of the nested content that they are signing. SAD Path Signatures provide this mechanism with the CESR SAD Path Language and new CESR attachment codes are detailed in this specification.

Streamable SADs

A primary goal of SAD Path Signatures is to allow any signed SAD to be streamed inline with any other CESR content. In support of that goal, SAD Path Signatures leverage CESR attachments to define a signature scheme that can be attached to any SAD content serialized as JSON [RFC4627], MessagePack [3] or CBOR [RFC8949]. Using this capability, SADs signed with SAD Path Signatures can be streamed inline in either the Text (T) or Binary (B) domain alongside any other KERI event Message over, for example TCP or UDP. In addition, signed SADs can be transported via HTTP as a CESR HTTP Request.

Nested Partial Signatures

SAD Path Signatures can be used to sign as many portions of a SAD as needed, including the entire SAD. The signed subsets are either SADs themselves or the SAID of a SAD that will be provided out of band. A new CESR Count Code is included with this specification to allow for multiple signatures on nested portions of a SAD to be grouped together under one attachment. By including a SAD Path in the new CESR attachment for grouping signatures, the entire group of signatures can be transposed across envelope boundaries by changing only the root path of the group attachment code.

Transposable Signature Attachments

There are several events in KERI that can contain context specific embedded SADs. Exchange events (`exn`) for peer-to-peer communication and Replay events (`rpy`) for responding to data requests as well as Expose events (`exp`) for providing anchored data are all examples of KERI events that contain embedded SADs as part of their payload. If the SAD payload for one of these event types is signed with a CESR attachment, the resulting structure is not embeddable in one of the serializations of map or dictionary like data models. (JSON, CBOR, MessagePack) supported by CESR. To solve this problem, SAD Path Signatures are transposable across envelope boundaries in that a single SAD signature or an entire signature group on any given SAD can be transposed to attach to the end of an enveloping SAD without losing its meaning. This unique feature is provided by the SAD Path language used in either a SAD signature or the root path designation in the outermost attachment code of any SAD signature group. These paths can be updated to point to the embedded location of the signed SAD inside the envelope. Protocols for verifiable credential issuance and proof presentation can be defined using

this capability to embed the same verifiable credential SAD at and location in an enveloping `exn` Message as appropriate for the protocol without having to define a unique signature scheme for each protocol.

SAD Path Language

SAD Path Signatures defines a SAD Path Language to be used in signature attachments for specifying the location of the SAD content within the signed SAD that a signature attachment is verifying. This path language has a more limited scope than alternatives like JSONPtr [RFC6901] or JSONPath JSONPath and is therefore simpler and more compact when encoding in CESR signature attachments. SADs in CESR and therefore SAD Path Signatures require static field ordering of all maps. The SAD path language takes advantage of this feature to allow for a Base64 compatible syntax into SADs even when a SAD uses non-Base64 compatible characters for field labels.

Description and Usage

The SAD path language contains a single reserved character, the `-` (dash) character. Similar to the `/` (forward slack) character in URLs, the `-` in the SAD Path Language is the path separator between components of the path. The `-` was selected because it is a one of the valid Base64 characters.

The simplest path in the SAD Path Language is a single `-` character representing the root path which specifies the top level of the SAD content.

Root Path

```
-
```

After the root path, path components follow, delimited by the `-` character. Path components may be integer indices into field labels or arrays or may be full field labels. No wildcards are supported by the SAD Path Language.

An example SAD Path using only labels that resolve to map contexts follows:

```
-a-personal
```

In addition, integers can be specified and their meaning is dependent on the context of the SAD.

```
-5-3-name
```

The rules for a SAD Path Language processor are simple. If a path consists of only a single `-`, it represents the root of the SAD and therefore the entire SAD content. Following any `-` character is a path component that points to a field if the current context is a map

in the SAD or is an index of an element if the current context is an array. It is an error for any sub-path to resolve to a value this is not a map or an array. Any trailing `-` character in a SAD Path can be ignored.

The root context (after the initial `-`) is always a map. Therefore, the first path component represents a field of that map. The SAD is traversed following the path components as field labels or indexes in arrays until the end of the path is reached. The value at the end of the path is then returned as the resolution of the SAD Path. If the current context is a map and the path component is an integer, the path component represents an index into fields of the map. This feature takes advantage of the static field ordering of SADs and is used against any SAD that contains field labels that use non-Base64 compatible characters or the `-` character. Any combination of integer and field label path components can be used when the current context is a map. All path components MUST be an integer when the current context is an array.

CESR Encoding for SAD Path Language

SAD Paths are variable raw size Primitives that require CESR variable size codes. The `A` small variable size code for SAD Paths will be used which has 3 code entries being added to the Master Code Table, `4A##`, `5A##` and `6A##` for SAD Paths with 0 lead bytes, 1 lead byte and 2 lead bytes respectively. This small variable size code is reserved for text values that only contain valid Base64 characters. The selector not only encodes the table but also implicitly encodes the number of lead bytes. The variable size is measured in quadlets of 4 characters each in the T domain and equivalently in triplets of 3 bytes each in the B domain. Thus, computing the number of characters when parsing or off-loading in the T domain means multiplying the variable size by 4. Computing the number of bytes when parsing or off-loading in the B domain means multiplying the variable size by 3. The two Base64 size characters provide value lengths in quadlets/triplets from 0 to 4095 (64**2 -1). This corresponds to path lengths of up to 16,380 characters (4095 * 4) or 12,285 bytes (4095 * 3).

SAD Path Examples

This section provides some more examples for SAD Path expressions. The examples are based on Authentic Chained Data Containers (ACDCs) representing Verifiable Credentials.

```
{
    "v": "ACDCCAACAAJSONAAIe.",
    "t": "acm",
    "d": "EO3117lnAbjDt66qe2PtgHooXKAYQT_C6SIbESMcJ5lN",
    "i": "EEDGM_DvZ9qFEAPf_FX08J3HX49ycrVvYVXe9isaP5SW",
    "s": "EGU_SHY-8ywNBJOqPKHr4sXV9tOtOwpYzYOM63_zUCDW",
    "a":
    {
        "d": "ED1wMKzV72L7YI1yJ3NXlClPUgvEerw4jRocOYxaZGtH",
```

```
        "i": "ECsGDKWAYtHBCkiDrzajkxs3Iw2g-dls3bLUsRP4yVdT",
        "dt": "2025-06-09T17:35:54.169967+00:00",
        "personal":
        {
            "name": "John Doe",
            "home": "Atlanta"
        },
        "p":
        [
            {
                "ref0":
                {
                    "name": "Amy",
                    "i": "ECmiMVHTfZIjhA_rovnfx73T3G_FJzIQtzDn1me
BVLAz"
                }
            },
            {
                "ref1":
                {
                    "name": "Bob",
                    "i": "ECWJZFBtllh99fESUOrBvT3EtBujWtDKCmyzDAX
WhYmf"
                }
            }
        ]
    }
}
```

Figure 1. Example ACDC Credential SAD

The examples in Table 1 represent all the features of the SAD Path language when referring to the SAD in Figure 1. along with the CESR text encoding.

SAD Path	CESR Encoding	Portion of SAD addressed by path
`-`	`'6AABAAA-'`	whole SAD
`-a-personal`	`4AADA-a-personal`	`{'name': 'John Doe', 'home': 'Atlanta'}`
`-5-3`	`4AAB-5-3`	`{'name': 'John Doe', 'home': 'Atlanta'}`
`-5-3-name`	`6AADAAA-5-3-name`	`'John Doe'`
`-a-personal-1`	`6AAEAAA-a-personal-1`	`'Atlanta'`
`-a-p-1-0`	`'4AAC-a-p-1-0`	`{'name': 'Bob', 'i': 'ECWJZFBtllh99fESUOrBvT3EtBujWtDKCmyzDAXWhYmf'}`
`-a-p-0-0-name`	`6AAEAAA-a-p-0-0-name`	`'Amy'`
`-a-p-0-ref0-i`	`6AAEAAA-a-p-0-ref0-i`	`'ECmiMVHTfZIjhA_rovnfx73T3G_FJzIQtzDn1meBVLAz'`

Alternative Pathing / Query Languages

The SAD Path language was chosen over alternatives such as JSONPtr and JSONPath in order to create a more compact representation of a pathing language in the text domain. Many of the features of the alternatives are not needed for SAD Path Signatures. The only token in the language (`-`) is Base64 compatible. The use of field indices in SADs (which require statically ordered fields) allows for Base64-compatible pathing even when the field labels of the target SAD are not Base64-compatible. The language achieves the goal of uniquely locating any path in a SAD using minimally sufficient means, allowing it to be embedded in a CESR attachment as Base64. Alternative syntaxes would need to be Base64 encoded to be used in a CESR attachment in the text domain, thus incurring the additional bandwidth cost of such an encoding.

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 designed for a future when they do. 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 consists of some cryptographic strength hashes. The analysis of D.J. Bernstein with regard to the collision resistance of cryptographic one-way hashing functions concludes that quantum computation provides no advantage over non-quantum techniques [PQCollisonCost].

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, it is uniquely and 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 public key's digest using non-quantum computation. Pre-quantum cryptographic strength is, therefore, not weakened post-quantum. A surprise quantum capability 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.

Bibliography

Normative section

1. KERI specification

2. ASCII, RFC20 https://www.rfc-editor.org/rfc/rfc20

3. MGPK specification MGPK

16. Reliable Asynchronous Event Transport, RAET, https://github.com/RaetProtocol/raet

18. RFC4627 The application/json Media Type for JavaScript Object Notation (JSON). D. Crockford; 2006-07. Status: Informational.

19. RFC4648 The Base16, Base32, and Base64 Data Encodings. S. Josefsson; 2006-10. Status: Proposed Standard.

20. RFC6901 JavaScript Object Notation (JSON) Pointer. P. Bryan, Ed.; K. Zyp; M. Nottingham, Ed.; 2013-04. Status: Proposed Standard.

21. RFC8949 Concise Binary Object Representation (CBOR). C. Bormann; P. Hoffman; 2020-12. Status: Internet Standard.

22. IETF RFC-2119 Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. 1997-03. Status: Best Current Practice

23. IETF RFC-8259 JSON. T. Bray, Ed. 2017-12. Status: Standards Track

24. Blake3 Specification Blake3. J. O'Connor; J-P. Aumasson; S. Neves ; Z. Wilcox-O'Hearn. Version 20211102173700.

Informative section

4. BOM, UTF Byte Order Mark, https://en.wikipedia.org/wiki/Byte_order_mark

5. DLog, Discrete Logarithm Problem, https://en.wikipedia.org/wiki/Discrete_logarithm

6. NaCL, https://nacl.cr.yp.to

7. MultiCodec Multiformats Codecs, MultiCodec

8. MultiCodec Table, MCTable

9. IPFS MultiFormats, IPFS https://richardschneider.github.io/net-ipfs-core/api/Ipfs.Registry.HashingAlgorithm.html

10. Base58Check Encoding, Base58Check

11. Wallet Import Format ECDSA Base58Check, WIF, Wallet Import Format

12. Binary to Text Encoding, Bin2Txt Binary to Text Encoding

13. UTF8, UTF-8 Unicode

14. Latin1 https://en.wikipedia.org/wiki/ISO/IEC_8859-1

15. Simple Text Oriented Messaging Protocol, STOMP, https://stomp.github.io

17. Analysis of the Effect of Core Affinity on High-Throughput Flows, Affinity, https://crd.lbl.gov/assets/Uploads/Nathan-NDM14.pdf

18. Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete?, D.J. Berstein, https://cr.yp.to/hash/collisioncost-20090517.pdf