TODO Working Group
Internet-Draft
Intended status: Informational
Expires: 2 June 2022

S. Smith
ProSapien LLC

29 November 2021

Composable Event Streaming Representation (CESR)
draft-ssmith-cesr-01

Abstract

The Composable Event Streaming Representation (CESR) is 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 separability of individual primtives. 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.

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/WebOfTrust/ietf-cesr.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 2 June 2022.

Table of Contents

1. Introduction

One way to better secure Internet communications is to use cryptographically verifiable primitives and data structures in communication. These provide essential building blocks for zero trust computing and networking architectures. Traditionally cryptographic primitives that include but are not limited to digests, salts, seeds (private keys), public keys, and digital signatures have

been largely represented in some type of 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, [JSON] documents, [RFC4627], system logs, audit logs, Ricardian contracts, and human readable text documents of many types.

Generic binary-to-text, [Bin2Txt], 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 [Base58Check] 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, [WIF]. But each application may use a non-interoperable encoding of type and optionally size. Interestingly because a binary encoding may include as a subset some codes that are in the text-printable compatible subset of [ASCII] such as ISO Latin-1, [Latin1] or UTF-8, [UTF8], one may _serendipitously_ find, for a given cryptographic primitive, a text-printable type code from a binary code table such as the table [MCTable] from [MultiCodec] for [IPFS]. Indeed some [Base58Check] applications take advantage of the binary MultiCodec tables but only used _serendipitous_ text compatible type codes. Serindipoudous text encodings that appear in binary code tables, do not, however, work in general for any size or type. So the serindipoudous approach is not universally applicable and is no substitute for a true textual encoding protocol for cryptographic primitives.

In general there is no standard text based encoding protocol that provides universal type, size, and value encoding for cryptographic primitives. Providing this capability is the primary motivation for the encoding protocol defined herein.

Importantly, a textual encoding that includes type, size, and value is self-framing. A self-framing text primitive may be parsed without needing any additional delimiting characters. Thus a stream of concatenated primitives may be individually parsed without the need to encapsulate the primitives inside textual delimiters or envelopes. Thus a textual self-framing encoding provides the core capability for a streaming text protocol like [STOMP] or [RAET]. Although a first

class textual encoding of cryptographic primitives is the primary motivation for the CESR protocol defined herein, 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 this protocol is generally useful to encode in text data data structures of all types not merely those that contain cryptographic primitives.

Textual encodings have numerous usability advantages over binary encodings. The one advantage, however, a binary encoding has over text is compactness. An encoding protocol that has the property we call _text-binary concatentation composability_ or more succinctly _composability_, enables both the usability of text and the compactness of binary. Composability may be the most uniquely innovative and useful feature of the encoding protocol defined herein.

1.1. Composability

Composability as defined here is 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. Essentially _composability_ provides round-trippable lossless conversion between text and binary 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 a stream of text primitives to binary for compact transmission that the stream processor at the other end may safely convert back to text en-masse for further processing or archival storage as text. With the addition of group framing codes as composable primitives, such a composable encoding protocol enables pipelining (multi-plexing and de-multiplexing) of 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 [Affinity].

1.2. Abstract Domain Representations

The cryptographic primitives defined here (i.e. CESR) inhabit three different domains each with a different representation. The first domain we call streamable text or _text_ and is denoted as _T_. The third domain we call streamable binary or _binary_ and is denoted as _B_. Composability is defined between the _T_ and _B_ domains. The third domain we call _raw_ 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)_. The _text code_ element of the _R_ domain pair is string of one or more text characters that provides the type and size information for the encoded primitive when in the _T_ domain. Actual use of cryptographic primitives happens in the _R_ domain using the _raw binary_ element of the (code, raw binary) 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.

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]. 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 indexed set of primitives in the _R_ domain is denoted with r[k].

1.2.1. Transformations Between Domains

Although, the composability property, mentioned above but described in detail below, 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 are also defined and supported by the protocol. As a result there is a total of six transformations, one in each direction between 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 we can complete of circuit of transformations that starts in any of the three domains and then crosses over the other two domains in either direction. For example starting in the _R_ domain we can traverse a circuit that crosses into the _T_ and _B_ domains and then crosses back into the _R_ domain as follows:

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

Likewise starting in the _R_ domain we can traverse a circuit that crosses into the _B_ and _T_ domains and then crosses back into the _R_ domain as follows:

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

1.2.2. 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 then we can denote the transformation 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 concatentation composability property 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 concatentation of the set of individually transformed primitives.

For example, suppose we have two primitives in the text domain, namely, t[0] and t[1] that each respectively transform 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 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 other primitives enables multiplexing and demultiplexing of arbitrary groups of primitives for pipelining and/or on or off loading of streams. The text domain provides usability and the binary domain provides compactness. Composability allows efficient conversion of composed (concatenated) groups of primitives without having to individually parse each primitive.

2. Concrete Domain Representations

Text, _T_, domain representations in CESR use only the characters from the the URL and filename safe variant of the IETF RFC-4648 Base64 standard, [RFC4648]. Unless otherwise indicated all references to Base64 (RFC-4648) 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 through Z, a through z, -, 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.

Base64, [RFC4648], by itself does not satisfy the composability property. In CESR, both _T_ and _B_ domain representations include a prepended framing code prefix that is structured in such a way as to ensure composability.

Suppose for example we wanted to use naive Base64 characters in the text domain and naive binary bytes in the binary domain. For the sake of the example we will call these naive text and naive binary encodings and domains. Recall that a byte encodes 8 bits of information and a Base64 character encodes 6 bits information. Furthermore suppose that we have three primitives denoted x, y, and z in the naive binary domain with lengths of 1, 2, and 3 bytes respectively.

In the following diagrams we denote each byte in a naive binary primitive with zero based most significant bit first indicies. For example, b1 is bit one b0 is bit zero and B0 for byte zero, B1 for byte 1, etc.

The byte and bit level diagram for x is shown below where we use X to denote its bytes:

```
|           X0          |
|b7:b6:b5:b4:b3:b2:b1:b0|
```

Likewise for y below:

```
   |           Y0          |           Y1          |
   |b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|
```

And finally for z below:

```
|           Z0          |           Z1          |           Z2          |
|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|
```

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 with zero based most significant character first indicies.

Therefore to encode x in Base64, for example, 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 is use a Base64 character where the non-coding bits are zeros. This is diagrammed as follows:

```
   |           X0          |
   |b7:b6:b5:b4:b3:b2:b1:b0|
   |        C0       |        C1       |
```

Naive Base64 encoding always pads each individual conversion of a string of bytes to an even multiple of four characters. This provides something that may be described as It may be described as a sort of one-way composability but it is not true composability because it only works for a set of distinct conversions that are concatented after conversion not a single conversion of a concatenated set. We show the pads by replacing the spaces with = characters. The naive Base64 conversion of x is as follows:

```
|           X0          |
|b7:b6:b5:b4:b3:b2:b1:b0|
|        C0       |        C1       |========C2=======|========C3=======|
```

Likewise y requires at least 3 Base64 characters to capture all of its 16 bits as follows:

```
   |           Y0          |           Y1          |
   |b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|
   |        C0       |        C1       |        C1       |
```

Alignment on a 4 character boundary requires one pad character this becomes:

```
|           Y0          |           Y1          |
|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|
|        C0       |        C1       |        C2       |========C3=======|
```

Finally because z requires exactly four Base64 characters to capture all of its 24 bits, there are no pad characters needed.

```
|           Z0          |           Z1          |           Z2          |
|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|
|        C0       |        C1       |        C2       |        C3       |
```

Suppose we concatenate x + y into a three byte composition in the naive binary domain before Base64 encoding the concatentated whole. We have the following:

```
|           X0          |           Y0          |           Y1          |
|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|b7:b6:b5:b4:b3:b2:b1:b0|
|        C0       |        C1       |        C2       |        C3       |
```

We see that the least significant two bits of X0 are encoded into the same character, C1 as the most significant four bits of Y0. Therefore, a text domain parser is unable to cleanly de-concatenate on a character by character basis the conversion of x + y into separate text domain primitives. Thus naive binary to Base64 conversion does not satisfy the composability constraint.

Suppose instead we start in the text domain with primitives u and v of lengths 1 and 3 characters respectively. If we concatenate these two primitives as u + v in text domain and then convert as a whole to naive binary. We have the following:

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

We see that all six bits of information in U0 is included with the least significant two bits of information in V0 in B0. 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. Thus naive Base64 to binary conversion does not satisfy the composability constraint.

Indeed the composability property is only satisfied if each primitive in the _T_ domain is an integer multiple of four Base64 characters and each primitive in the _B_ domain is an integer multiple of three bytes. Each of Four Base64 characters and three bytes capture twenty-four bits of information. Twenty-four is the least common multiple of six and eight. Therefore primitive lengths that integer

multiples of either four Base64 characters or three bytes in their respective domains capture integer multiples of twenty-four bits of information. Given that constraint, conversion of concatenated primitives in one domain never result in two adjacent primitives sharing the same byte or character in the converted domain.

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 twenty-four bit boundaries to satisfy the composibility property. This means that the minimum length of any primitive in the B domain is three bytes and the minimum length of any primitive in the T domain is four Base64 characters.

2.1. Stable Text Type Codes

There are many coding schemes that could satisfy the composability constraint of alignment on 24 bit boundaries. The main reason for using a _T_ domain centric encoding is higher usability or human friendliness. Indeed a primary design goal of CESR is to select an encoding approach that provides such high usability or human friendliness in the _T_ domain. This type of usability goal is simpley not realizable in the _B_ domain. The B domain's purpose is merely to provide convenient compactness at scale. We believe usability in the _T_ domain is maximized when the type portion of the prepended framing code is _stable_ or _invariant_. 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 that have fixed lengths, i.e. all primitives of that type have the same length, stable type coding aids not only visual type but visual size recognition.

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 integer 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 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 no not. By in large 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.

2.2. Code Characters and Ante Bytes

There are two ways to provide the required alignment on 24 bit boundaries to satisfy the composability property. One is to increase the size of text code to ensure that the _T_ domain primitive has a total size (length) that is an integer multiple of 4. The other is to increase the size of the raw binary value by pre-pending pad bytes of zeros to the raw binary value before conversion to Base64 to ensure the total size of the raw binary value with pre-pended bytes is an integer multiple of 3 bytes. This ensures that size in characters of the Base64 conversion of the pre-padded raw binary is an integer multiple of 4 characters. In this case the length of the pre-pended type code MUST also therefore be an integer multiple of 4 characters so that the total length of the _T_ domain primitive with code is an integer multiple of 4 characters.

The first way may be more compact in some cases. The second way may be easier to compute in some cases. In order to avoid confusion with the use of the term pad character, when pre-padding with bytes we use the term ante bytes. The term pad may be confusing not merely

because both ways use a type of padding but it is also true that the the number of pad characters when padding post-conversion equals the number of ante bytes when padding pre-conversion.

Suppose for example the raw binary value is 32 bytes in length. The next higher integer multiple of 3 is 33 bytes. Thus 1 additional ante byte is needed to make the size (length in byte) of raw binary an integer multiple of 3. The 1 ante 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 we convert a 32 byte raw binary value to Base64 the converted value will have 1 pad character which may be replaced with a text code character. In both cases the resultant length in Base64 is 44 characters.

Similarly, a 64 byte sized raw binary needs 2 ante 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 we can use pre-conversion ante bytes or post-conversion pad characters in our coding scheme to ensure composable 24 bit alignment.

2.3. 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. We believe this is best achieved with multiple code tables each with a different coding scheme that is 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 primitive.

3. Text Coding Scheme Design

3.1. Text Code Size

Recall from above, 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 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 converting the _raw binary_ element to Base64, then stripping off any Base64 pad characters then finally prepending the text code element to the result of the conversion.

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 pad characters, a set of Base64 strings resulting from the subsequent conversions of a set of binary strings could be concatenated and then converted back to binary en masse while preserving byte boundaries. In order to preserve byte boundaries the pad characters MUST be stripped in the conversion. Because the pad characters are stripped, this approach does not provide two-way or true composability as defined above. The number of pad characters is a function of the length of the binary string. Let _N_ be the length

the string. When N mod 3 = 1 then there are 8 bits in remainder that must be encoded into Base64. Recall the examples above, a single byte (8 bits) require two Base64 characters. The first encodes 6 bits and the second 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. When N mod 3 = 2 then 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 pad character is required to pad out the resulting conversion so that is length is an integer multiple of 4 characters. When N mod 3 = 0 then the binary string is aligned on a 24 bit boundary and no pad characters are required to ensure the length of the Base64 conversion is an integer multiple of 4 characters.

The number of requred Base64 pad characters to ensure that a given conversion to Base64 has a length that is an integer multiple of 4 may be computed with the following formula:

ps = (3 - (N mod 3)) mod 3), where ps is the pad size 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 appropirate 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).

This 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 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 is a function of the equivalent pad size determined by the length N mod 3 of the raw binary value. If we let _M_ be a non-negative integer valued variable then we have three cases:

Pad Size	Code Size
0	4M

1	4M+1
2	4M+2

Table 1

The minimum code sizes are 1, 2, and 4 characters for pad sizes of 1, 2, and 0 characters with _M_ equal 0, 0, and 1 respectively. By increasing _M_ we can have larger code sizes for a given pad size.

3.2. Count, Group, or Framing Codes

As mentioned above one of the primary advantages of a composable encoding is that special framing codes can be specified to support groups of primitivies. Grouping enables pipelining. Other suitable terms for these special framing codes are _group codes_ or _count codes_. These are suitable because they can be used to count characters, primitives in a group, or count groups of primitives in a larger group. We can also use count codes as separators to organize a stream of primitives or to interleave non-native serializations. A count code is its own primitive. But it is a primitive that does not include a raw binary value, only the text code. Because a count code's raw binary element is empty, its pad size is always 0. Thus a count code's size is always an integer multiple of 4 characters i.e. 4, 8, etc.

3.3. Interleaved Non-CESR Serializations

One extremely useful property of CESR is that special count codes enable CESR to be interleaved with other serializations. For example, Many applications use JSON JSON [RFC4627], CBOR CBOR [RFC8949], or MsgPack (MGPK) [MGPK] to serialize flexible self-describing data structures based on hash maps, also know as dictionaries. With respect to hash map serializations, CESR primitives may appear in two different contexts. The first context is as a delimited text primitive inside of a hash map serialization. The delimited text may be either the key or value of a (key, value) pair. The second context is as a standalone serialization that is interleaved with hash map serializations in a stream. Special CESR count codes enable support for the second context of interleaving standalone CESR with other serializations.

3.4. 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. Better yet is 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 cold start. Good cold start re-synchronization is essential to robust performant stream processing.

For example, in 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 UDP each packet is individually framed 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

3.4.1. Performant Resynchronization with Unique Start Bits

Given the popularity of three specific serializations, namely, JSON, CBOR, and MGPK, more fine grained serialization boundary detection for interleaving CESR may be highly beneficial both from a performation and robustness perspective. 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 (framing) codes in both the _T_ domain (Base64) and the _B_ domain should be unique. This would provide the analogous equivalent of a UTF Byte Order Mark (BOM) [BOM]. A BOM enables a parser of UTF encoded documents to determine if the UTF codes are big endian or little endian [[19]]. In the CESR case this feature would enable a stream parser to know if a count code along with its associated counted or framed group of primitives are expressed in the _T_ or _B_ domain. Toghether these impose the constraint that the boundary start bits for interleaved text CESR, binary CESR, JSON, CBOR, and MGPK be mutually distinct.

Amongst the codes for map objects in the JSON, CBOR, and MGPK only the first three bits 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 the serialized mapping object are 0b101. In MGPK (MsgPack) there are three different mapping object codes. The _FixMap_ code starts with 0b100. Both the _Map16_ code and _Map32_ code start with 0b110.

So we have the set of four used starting tritets (3 bits) in numeric order of 0b011, 0b100, 0b101, and 0b110. This leaves four unused tritets, namely, 0b000, 0b001, 0b010, and 0b111 that may be selected as the CESR count (framing) code start bits. In Base64 there are two codes that satisfy our 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 the 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

This gives us two different Base64 characters, - and _ we can use for the first character of any framing (count) code in the _T_ domain. This also means we can have two different classes of framing (count)

codes. This also provides a BOM like capability to determine if a framing code is expressed in the _T_ or _B_ domain. To clarify, if a stream starts with the tritet 0b111 then the stream is _B_ domain CESR and a stream parser would thereby know how to convert the first sextet of the stream to determine which of the two framing codes is being used, 0x3E or ox3F . If on the other hand the framing code starts with either of the tritets 0b001 or 0b010 then the framing code is expressed in the _T_ domain and a stream parser likewise

would thereby know how to convert the first character (octet) of the framing code to determine which framing code is being used. Otherwise if a stream starts with 0b100 then is JSON, with 0b101 then its CBOR and with either 0b011,and 0b110 then its MGPK.

This is summaraized in the following table:

Starting Tritet	Serialization	Character
0b000		
0b001	CESR _T_ Domain Count (Group) Code	-
0b010	CESR _T_ Domain Op Code	_
0b011	JSON	{
0b100	MGPK	
0b101	CBOR	
0b110	MGPK	
0b111	CESR _B_ Domain	

Table 2

3.4.2. Stream Parsing Rules

Given this set of tritets (3 bits) we can express a requirement for well formed stream start and restart.

Each stream MUST start (restart) with one of five tritets:

1) A framing count (group) code in CESR _T_ domain 2) A framing count (group) code in CESR _B_ Domain. 3) A JSON encoded mapping. 4) A CBOR encoded Mapping. 5) A MGPK encoded mapping.

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 five it is. When the first tritet is a framing code then, the remainder of framing code itself will include the additional information needed to parse the attached group. When the first tritet indicates its JSON, CBOR, or MGPK, then the mapping's first field must be a version string that provides the additional information needed to fully parse the associated encoded serialization.

The stream MUST resume with a starting byte that starts with one of the 5 tritets, either another framing code expressed in the _T_ or _B_ domain or a new JSON, CBOR, or MGPK encoded mapping.

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

3.5. Compact Fixed Size Codes

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. 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 a minimally sufficient cryptographic strength and more cryptographic strength is just wasting computation and bandwidth. Cryptographic strength is measured in bits of entropy which also corresponds to the number trials that must 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 [NaCL] 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 [DLog]). Thus 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 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. Indeed the sweet spots for modern cryptographic raw primitive lengths are 32 bytes for many digests as well as EdDSA public keys and 64 bytes 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 for 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, then the minimium size text code is 4 characters. So the minimally sized texts code tables are 1, 2, and 4 characters respectively.

Given that a given cryptographic primitive type has a known fixed raw binary size then we can efficiently encode that primitive type and size with just the type information. The size is given by the type.

So 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.

3.6. Code Table Selectors

In order to efficiently parse a stream of primitives with types from multiple text code tables the first character in the text code must be a code table selector character. Thus the 1 character text code table must do double duty. It must provide selectors for the different text code tables and also provide type codes for the most popular primitives that have a pad size of 1. There are 64 Base64 characters (64 values). We only need 12 tables to support all the codes and code formats needed for the foreseeable future. Therefore only 12 of those characters need be dedicated as code table selectors that leaves 52 characters that may be used for 1 character type

codes. This gives a total of 13 type code tables consisting of the dual purpose 1 character selector table and 12 other tables.

As described above the selector characters for the framing or count code tables that best support interleaved JSON, CBOR, and MGPK are - and _. We use the numerals 0 through 9 to each serve as a selector. That leaves the letters A through Z and a through 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 is 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.

3.7. Small Fixed 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 text code size equals the number of pad characters, i.e. the pad size.

3.7.1. One Character Fixed Raw Size Table

The one character type code table does not have 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 pad size of 1.

3.7.2. Two Character Fixed Raw Size Table

The two character type code table uses 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.

3.8. Large Fixed Raw Size Tables

The three tables in this group are for large fixed raw size primitives. These three tables use 0, 1 or 2 ante 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 not only encodes the table but also implicitly encodes the number of ante bytes. 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.

3.8.1. Large Fixed Raw Size Table With 0 Ante Bytes

This table uses 1 as its first character or selector. The remaining 3 characters provide the types codes. Only fixed size raw binaries with 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 pad size of 0.

3.8.2. Large Fixed Raw Size Table With 1 Ante Byte

This table uses 2 as its first character or selector. The remaining 3 characters provide the types codes. Only fixed size raw binaries with 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 pad size of 1.

3.8.3. Large Fixed Raw Size Table With 1 Ante Byte

This table uses 3 as its first character or selector. The remaining 3 characters provide the types codes. Only fixed size raw binaries with 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 pad size of 2.

3.9. Small Variable 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 sizing such as encrypted material. Indeed CESR is meant to support not only cryptographic material types but other basic types such as generic text strings. These 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 ante 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 is 64. A given type code is repeated in each table for the same type. What is different for each table is the number of ante bytes. The selector not only encodes the table but also implicitly encodes the number of ante 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).

3.9.1. Small Variable Raw Size Table With 0 Ante 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 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 pad size of 0 that each include 0 ante bytes.

3.9.2. Small Variable Raw Size Table With 1 Ante 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 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 pad size of 1 that each include 1 ante byte.

3.9.3. Small Variable Raw Size Table With 2 Ante 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 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 int the _B_ domain. All are raw binary primitives with pad size of 2 that each include 2 ante bytes.

3.10. Large Variable Raw Size Tables

Many legacy cryptographic libraries such as OpenSSL and GPG support any sized variable sized primitive for keys, signatures and digests. 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 raw primitives for the foreseeable future.

The three tables in this group are for large variable raw size primitives. These three tables use 0, 1 or 2 ante 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. 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 ante bytes. The selector not only encodes the table but also implicitly encodes the number of

ante 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).

3.10.1. Large Variable Raw Size Table With 0 Ante 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 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 include 0 ante bytes.

3.10.2. Large Variable Raw Size Table With 1 Ante 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 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 pad size of 1 that each include 1 ante bytes.

3.10.3. Large Variable Raw Size Table With 2 Ante 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 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 pad size of 2 that each include 2 ante bytes.

3.11. Count (Framing) Code Tables

There may be as many at 13 count code tables, but only two are currently specified. These two are the small count, four character table and the large count, eight character table. Because count codes only count quadlets/triplets, primitives or groups of primitives, count codes have no value component, but only type and size components. Because primitives are already guaranteed to be composable count codes do not need to account for pad size as long as the count code itself is aligned on a 24 bit boundary. The count code type indicates the type of primitive being counted and the size indicates how many of that type. Both count code tables use the first two characters as a nested set of selectors. The first selector uses- as the initial selector for count codes. The next character is either a selector for another count code table or is the type for the small count code table. When the second character is numeral 0 - 9 or the letters - or _ then it is a secondary cound code table selector. When the second character is a letter in the range A - Z or a - z then it is a unique count code type. This given a total of 52 single character count code types.

3.11.1. Small Count Code Table

Codes in the small count code table are each four characters long. The first character is the selector -. The second character is the count code type. the last two characters are the count size as a Base64 encoded integer. The count code type MUST be a letter A - Z or a - z. If the second character is not a letter but is a numeral 0

- 9 or - or _ then it is a selector for a different count code table. The set of letters provide 52 unique count codes. A two character size provides counts from 0 to 4095 (64**2 - 1).

3.11.2. Large Count Code Table

Codes in the large count code table are each 8 characters long. The first two characters are the selectors -0. The next two characters are the count code type. the last four characters are the count size as a Base64 encoded integer. With two characters for type, there are 4096 unique large count code types. A four character size provides counts from 0 to 16,777,215 (64**4 - 1).

3.12. Op Code Tables

The _ selector is reserved for the yet to be defined op code table or tables. Op codes are meant to provide stream processing instructions that are more general and flexible than simply concatenated primitives or groups of primitives.

3.13. Selector Codes and Encoding Schemes

The following table summarizes the _T_ domain coding schemes for the 13 code tables defined above.

Selector	Selector	Type Chars	Value Size Chars	Code Size	Ante Bytes	Pad Size	Format
[A-Z,a-z]		1*	0	1	0	1	$&&&
0		1	0	2	0	2	0$&&
1		3	0	4	0	0	1$$$&&&&
2		3	0	4	1	1	2$$$&&&&
3		3	0	4	2	2	3$$$&&&&

4		1	2	4	0	0	4$##&&&&
5		1	2	4	1	1	5$##&&&&
6		1	2	4	2	2	6$##&&&&
7		3	4	8	0	0	7$$$####&&&&
8		3	4	8	1	1	8$$$####&&&&
9		3	4	8	2	2	9$$$####&&&&
-	[A-Z,a-z]	1*	0	4	0	0	-$##
-	0	2	0	8	0	0	-0$$####
_		TBD	TBD	TBD	TBD	TBD	_

Table 3

* selector character is also type character

Character format symbol definitions:
$ means type code character from subset of Base64 [A-Z,a-z,0-9,-,_].
# means a Base64 digit as part of a base 64 integer that determines

the number of following quadlets or triplets in the primitive or when part of a count code, the count of following primitives or groups of primitives.
& represents one or more Base64 value characters representing the converted raw binary value included ante bytes when applicable. The actual number of chars is determined by the prep-ended text code.
TBD means to be determined

3.14. Parse Size Table

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 parse then extracts hard size characters from the text stream. These characters

form an index in to 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 domian 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.

selector	hs
B	1
0	2
5	2

Table 4

hard sized index	hs	ss	vs	fs	as	ps
B	1	0	43*	44	0	1
0B	2	0	86*	88	0	2*

5A	2	2	#	#	1	1*

Table 5

* size may be calculated from other sizes.
# size may be calculated from extracted code characters given by other sizes.

_hs_ means hard size in chars.
_ss_ means soft size in chars.
_cs_ means code size where _cs = hs + ss_.
_vs_ means value size in chars.
_fs_ means full size in chars where _fs = hs + ss + vs_.
_as_ means ante size in bytes.
_ps_ means pad size in chars.
_rs_ means raw size in bytes of binary value.
_as_ means ante size in bytes.
_bs_ means binary size in bytes where _bs = as + rs_.

3.15. Special Context Specific Code Tables

The table above that provides the encoding schemes each with an associated code table that provides the type codes or set of codes for each associated primitive type. These coding schemes constitute the basic set of code tables. This basic set 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 code tables. Currently there is only one context specific coding scheme, that is, for indexed signatures. A common use case are thresholded multi-signature schemes. A threshold satisficing subset of signatures belonging to an ordered or list of public keys may be provided as part of stream of primitives. One way to compactly associated each signature with its public key is to include in the text code for that signature the index into the ordered set of public keys. The typical raw binary size for signatures is 64 bytes which has 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 Base64 encoded integer index. By using a similar dual selector type code character scheme as above, where 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 supports up to 64 members in the public key list. A selector can be used to select a large text code with more characters dedicated to larger indicies. Current only a small table is defined.

A new signature scheme based on Ed448 with 114 byte signatures signatures is also supported. These signatures have a pad size of zero so require a four charactor text code. The first characters is the selector 0, the second characters is the type with 64 values, the last two characters provide the index as a Base64 encoded integer with 4096 different values.

The associate indexed schemes are provided in the following table.

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

Table 6

* selector character is also type character

Character format symbol definitions:
$ means type code character from subset of Base64 [A-Z,a-z,0-9,-,_].
# means a Base64 digit as part of a base 64 integer that determines the index.
& represents one or more Base64 value characters representing the converted raw binary value included ante bytes when applicable. The actual number of chars is determined by the prep-ended text code.
TBD means to be determined

4. Master Code Table

4.1. Filling Code Table

The approach to filling the tables is a first needed first served basis. In addition the requirement that all cryptographic operations maintain at least 128 bits of cryptographic strength precludes the entry of many weak cryptographic suites into the compact tables. CESR's compact code table includes only best-of-class cryptographic operations. In 2022 it is expected that NIST will approve standardized post-quantum resistant cryptographic signatures at which time codes for the most appropriate post quantume signature suites will be added. Falcon appears to be the leader with open source code already available.

4.2. Description

This master table includes all three types of codes separated by headers. The 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 index or count portion of the code 5) The length in characters of the fully qualified primitive including code and append material or number of elements in group.

Code	Description	Code Length	Count or Index Length	Total Length
	Basic One Character Codes			
A	Random seed of Ed25519 private key of length 256 bits	1		44
B	Ed25519 non-transferable prefix public signing verification key. Basic derivation.	1		44

C	X25519 public encryption key. May be converted from Ed25519 public signing verification key.	1			44

D	Ed25519 public signing verification key. Basic derivation.	1			44
E	Blake3-256 Digest. Self-addressing derivation.	1			44
F	Blake2b-256 Digest. Self-addressing derivation.	1			44
G	Blake2s-256 Digest. Self-addressing derivation.	1			44
H	SHA3-256 Digest. Self-addressing derivation.	1			44
I	SHA2-256 Digest. Self-addressing derivation.	1			44
J	Random seed of ECDSA secp256k1 private key of length 256 bits	1			44
K	Random seed of Ed448 private key of length 448 bits	1			76
L	X448 public encryption key. May be converted from Ed448 public signing verification key.	1			76

M	Short value of length 16 bits	1		4	
	Basic Two Character Codes				
0A	Random salt, seed, private key, or sequence number of length 128 bits	2		24	

0B	Ed25519 signature. Self-signing derivation.	2		88	
0C	ECDSA secp256k1 signature. Self-signing derivation.	2		88	
0D	Blake3-512 Digest. Self-addressing derivation.	2		88	
0E	Blake2b-512 Digest. Self-addressing derivation.	2		88	
0F	SHA3-512 Digest. Self-addressing derivation.	2		88	
0G	SHA2-512 Digest. Self-addressing derivation.	2		88	
0H	Long value of length 32 bits	2		8	
	Basic Four Character Codes				
1AAA	ECDSA secp256k1 non-transferable prefix	4		48	

	public signing verification key. Basic derivation.				
1AAB	ECDSA secp256k1 public signing verification or encryption key. Basic derivation.	4		48	
1AAC	Ed448 non-transferable prefix public signing verification key. Basic derivation.	4		80	
1AAD	Ed448 public signing verification key. Basic derivation.	4		80	
1AAE	Ed448 signature. Self-signing derivation.	4		156	
1AAF	Tag Base64 4 chars or 3 byte number	4		8	
1AAG	DateTime Base64 custom encoded 32 char ISO-8601 DateTime	4		36	
	Indexed Two Character Codes				
A#	Ed25519 indexed signature	2	1	88	
B#	ECDSA secp256k1 indexed signature	2	1	88	
	Indexed Four Character Codes				
0A##	Ed448 indexed signature	4	2	156	

0B##	Label Base64 chars of variable length L=N*4 where N is value of index total = L+4	4	2	Variable
	Counter Four Character Codes			
-A##	Count of attached qualified Base64 indexed controller signatures	4	2	4
-B##	Count of attached qualified Base64 indexed witness signatures	4	2	4
-C##	Count of attached qualified Base64 nontransferable identifier receipt couples pre+sig	4	2	4
-D##	Count of attached qualified Base64 transferable identifier	4	2	4

	receipt quadruples pre+snu+dig+sig			
-E##	Count of attached qualified Base64 first seen replay couples fn+dt	4	2	4
-F##	Count of attached qualified Base64 transferable indexed sig groups pre+snu+dig + idx sig group	4	2	4

-U##	Count of qualified Base64 groups or primitives in message data	4	2	4	
-V##	Count of total attached grouped material qualified Base64 4 char quadlets	4	2	4	
-W##	Count of total message data grouped material qualified Base64 4 char quadlets	4	2	4	
-X##	Count of total group message data plus attachments qualified Base64 4 char quadlets	4	2	4	
-Y##	Count of qualified Base64 groups or primitives in group. (context dependent)	4	2	4	
-Z##	Count of grouped material qualified Base64 4 char quadlets (context dependent)	4	2	4	
-a##	Count of anchor seal groups in list (anchor seal list) (a)	4	2	4	
-c##	Count of config traits (each trait is 4 char quadlet (configuration trait list) (c)	4	2	4	
-d##	Count of digest seal	4	2	4	

	Base64 4 char quadlets in digest (digest seal (d)				
-e##	Count of event seal Base64 4 char quadlets in seal triple of (event seal) (i, s, d)	4	2	4	
-k##	Count of keys in list (key list) (k)	4	2	4	
-l##	Count of locations seal Base64 4 char quadlets in seal quadruple of (location seal) (i, s, t, p)	4	2	4	
-r##	Count of root digest seal Base64 4 char quadlets in root digest (root digest) (rd)	4	2	4	
-w##	Count of witnesses in list (witness list or witness remove list or witness add list) (w, wr, wa)	4	2	4	
	Counter Eight Character Codes				
-0U#####	Count of qualified Base64 groups or primitives in message data	8	5	8	
-0V#####	Count of total attached grouped material qualified Base64 4 char quadlets	8	5	8	

-0W#####	Count of total message data grouped material qualified Base64 4 char quadlets	8	5	8
-0X#####	Count of total group message data plus attachments qualified Base64 4 char quadlets	8	5	8
-0Y#####	Count of qualified Base64 groups or primitives in group (context dependent)	8	5	8
-0Z#####	Count of grouped material qualified Base64 4 char quadlets (context dependent)	8	5	8
-0a#####	Count of anchor seals (seal groups in list)	8	5	8

Table 7

The table includes complex groups that are composed of other groups. For example consider the counter attachment group with code-F## 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 more more attached groups where each group consists of a triple pre+snu+dig followed by a ControllerIdxSigs group that in turn consists of a counter code -A## followed by one or more indexed signature primitives. The following example details how this complex group may appear.

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

```
-FAB     # Trans Indexed Sig Groups counter code 1 following group
E_T2_p83_gRSuAYvGhqV3S0JzYEF2dIa-OCPLbIhBO7Y    # trans prefix of signer for si
-EAB0AAAAAAAAAAAAAAAAAAAAAB    # sequence number of est event of signer's publ
EwmQtlcszNoEIDfqD-Zih3N6o5B3humRKvBBln2juTEM      # digest of est event of sign
-AAD     # Controller Indexed Sigs counter code 3 following sigs
AA5267UlFg1jHee4Dauht77SzGl8WUC_0oimYG5If3SdIOSzWM8Qs9SFajAilQcozXJVnbkY5stG_K4
ABBgeqntZW3Gu4HL0h3odYz6LaZ_SMfmITL-Btoq_7OZFe3L16jmOe49Ur108wH7mnBaq2E_0U0N0c5
ACTD7NDX93ZGTkZBBuSeSGsAQ7u0hngpNTZTK_Um7rUZGnLRNJvo5oOnnC1J2iBQHuxoq8PyjdT3BHS
```

5. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

6. Security Considerations

TODO Security

7. IANA Considerations

This document has no IANA actions.

8. References

8.1. Normative References

[RFC20] "ASCII format for network interchange", 29 July 2020, <https://datatracker.ietf.org/doc/rfc20/>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", 21 January 2020, <https://datatracker.ietf.org/doc/rfc4648/>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

8.2. Informative References

[Affinity] "Analysis of the Effect of Core Affinity on High-Throughput Flows", 16 November 2014, <https://crd.lbl.gov/assets/Uploads/Nathan-NDM14.pdf>.

[ASCII] "Text Printable ASCII Characters", n.d., <https://en.wikipedia.org/wiki/ASCII>.

[Base58Check] "Base58Check Encoding", n.d., <https://en.bitcoin.it/wiki/Base58Check_encoding>.

[Bin2Txt] "Binary to Text Encoding", n.d., <https://en.wikipedia.org/wiki/Binary-to-text_encoding>.

[BOM] "UTF Byte Order Mark", n.d., <https://en.wikipedia.org/wiki/Byte_order_mark>.

[CBOR] "CBOR Mapping Object Codes", n.d., <https://en.wikipedia.org/wiki/CBOR>.

[DLog] "Discrete Logarithm Problem", n.d., <https://en.wikipedia.org/wiki/Discrete_logarithm>.

[IPFS] "IPFS MultiFormats", n.d., <https://richardschneider.github.io/net-ipfs-core/api/Ipfs.Registry.HashingAlgorithm.html>.

[JSON] "JavaScript Object Notation Delimeters", n.d., <https://www.json.org/json-en.html>.

[KERI] Smith, S., "Key Event Receipt Infrastructure (KERI)", 2021, <https://arxiv.org/abs/1907.02143>.

[Latin1] "Latin-1 ISO 8859-1", n.d., <https://en.wikipedia.org/wiki/ISO/IEC_8859-1>.

[MCTable] "MultiCodec Table", n.d., <https://github.com/multiformats/multicodec/blob/master/table.csv>.

[MGPK] "Msgpack Mapping Object Codes", n.d., <https://github.com/msgpack/msgpack/blob/master/spec.md>.

[MultiCodec] "MultiCodec Multiformats Codecs", n.d., <https://github.com/multiformats/multicodec>.

[NaCL] "NaCl Networking and Cryptography library", n.d., <https://nacl.cr.yp.to>.

[RAET] "Reliable Asynchronous Event Transport", n.d., <https://github.com/RaetProtocol/raet>.

[RFC4627] "The application/json Media Type for JavaScript Object Notation (JSON)", n.d., <https://datatracker.ietf.org/doc/rfc4627/>.

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", 4 December 2020, <https://datatracker.ietf.org/doc/rfc8949/>.

[STOMP] "Simple Text Oriented Messaging Protocol", n.d., <https://stomp.github.io>.

[UTF8] "UTF-8 Unicode", n.d., <https://en.wikipedia.org/wiki/UTF-8>.

[WIF] "Wallet Import Format ECDSA Base58Check", n.d., <https://en.bitcoin.it/wiki/Wallet_import_format>.

Acknowledgments

The keripy development team, the KERI community and the ToIP ACDC working group.

Author's Address

S. Smith
ProSapien LLC

Email: sam@prosapien.com