HAL
open science

Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Schemes for FECFRAME (RFC 8681)

Vincent Roca, Belkacem Teibi

To cite this version:

Vincent Roca, Belkacem Teibi. Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Schemes for FECFRAME (RFC 8681). 2020. hal-01630089v4

HAL Id: hal-01630089

https://inria.hal.science/hal-01630089v4

Submitted on 22 Jan 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Stream: Internet Engineering Task Force (IETF)
RFC: 8681
Category: Standards Track
Published: January 2020
ISSN: 2070-1721
Authors: V. Roca, *INRIA*; B. Teibi, *INRIA*

RFC 8681
Sliding Window Random Linear Code (RLC) Forward Erasure Correction (FEC) Schemes for FECFRAME

Abstract

This document describes two fully specified Forward Erasure Correction (FEC) Schemes for Sliding Window Random Linear Codes (RLC), one for RLC over the Galois Field (a.k.a., Finite Field) GF(2), a second one for RLC over the Galois Field GF(2^8), each time with the possibility of controlling the code density. They can protect arbitrary media streams along the lines defined by FECFRAME extended to Sliding Window FEC Codes. These Sliding Window FEC Codes rely on an encoding window that slides over the source symbols, generating new repair symbols whenever needed. Compared to block FEC codes, these Sliding Window FEC Codes offer key advantages with real-time flows in terms of reduced FEC-related latency while often providing improved packet erasure recovery capabilities.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc8681.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction

1.1. Limits of Block Codes with Real-Time Flows

1.2. Lower Latency and Better Protection of Real-Time Flows with the Sliding Window RLC Codes

1.3. Small Transmission Overheads with the Sliding Window RLC FEC Scheme

1.4. Document Organization

2. Definitions and Abbreviations

3. Common Procedures

3.1. Codec Parameters

3.2. ADU, ADUI, and Source Symbols Mappings

3.3. Encoding Window Management

3.4. Source Symbol Identification

3.5. Pseudorandom Number Generator (PRNG)

3.6. Coding Coefficients Generation Function

3.7. Finite Field Operations

3.7.1. Finite Field Definitions

3.7.2. Linear Combination of Source Symbol Computation

4. Sliding Window RLC FEC Scheme over GF(2^8) for Arbitrary Packet Flows

4.1. Formats and Codes

4.1.1. FEC Framework Configuration Information

4.1.2. Explicit Source FEC Payload ID

4.1.3. Repair FEC Payload ID

4.2. Procedures

5. Sliding Window RLC FEC Scheme over GF(2) for Arbitrary Packet Flows

5.1. Formats and Codes

5.1.1. FEC Framework Configuration Information

5.1.2. Explicit Source FEC Payload ID

5.1.3. Repair FEC Payload ID

5.2. Procedures

6. FEC Code Specification

6.1. Encoding Side

6.2. Decoding Side

7. Security Considerations

7.1. Attacks Against the Data Flow

7.1.1. Access to Confidential Content

7.1.2. Content Corruption

7.2. Attacks Against the FEC Parameters

7.3. When Several Source Flows are to be Protected Together

7.4. Baseline Secure FEC Framework Operation

7.5. Additional Security Considerations for Numerical Computations

8. Operations and Management Considerations

8.1. Operational Recommendations: Finite Field GF(2) Versus GF(2^8)

8.2. Operational Recommendations: Coding Coefficients Density Threshold

9. IANA Considerations

10. References

10.1. Normative References

10.2. Informative References

Appendix A. TinyMT32 Validation Criteria (Normative)

Appendix B. Assessing the PRNG Adequacy (Informational)

Appendix C. Possible Parameter Derivation (Informational)

C.1. Case of a CBR Real-Time Flow

C.2. Other Types of Real-Time Flow

C.3. Case of a Non-Real-Time Flow

Appendix D. Decoding Beyond Maximum Latency Optimization (Informational)

Acknowledgments

Authors' Addresses

1. Introduction

Application-Level Forward Erasure Correction (AL-FEC) codes, or simply FEC codes, are a key element of communication systems. They are used to recover from packet losses (or erasures) during content delivery sessions to a potentially large number of receivers (multicast/broadcast transmissions). This is the case with the File Delivery over Unidirectional Transport (FLUTE)/ Asynchronous Layered Coding (ALC) protocol [RFC6726] when used for reliable file transfers over lossy networks, and the FECFRAME protocol [RFC6363] when used for reliable continuous media transfers over lossy networks.

The present document only focuses on the FECFRAME protocol, which is used in multicast/ broadcast delivery mode, particularly for content that features stringent real-time constraints: each source packet has a maximum validity period after which it will not be considered by the destination application.

1.1. Limits of Block Codes with Real-Time Flows

With FECFRAME, there is a single FEC encoding point (either an end host/server (source) or a middlebox) and a single FEC decoding point per receiver (either an end host (receiver) or middlebox). In this context, currently standardized AL-FEC codes for FECFRAME like Reed-Solomon [RFC6865], LDPC-Staircase [RFC6816], or Raptor/RaptorQ [RFC6681], are all linear block codes: they require the data flow to be segmented into blocks of a predefined maximum size.

To define this block size, it is required to find an appropriate balance between robustness and decoding latency: the larger the block size, the higher the robustness (e.g., in case of long packet erasure bursts), but also the higher the maximum decoding latency (i.e., the maximum time required to recover a lost (erased) packet thanks to FEC protection). Therefore, with a multicast/ broadcast session where different receivers experience different packet loss rates, the block size should be chosen by considering the worst communication conditions one wants to support, but without exceeding the desired maximum decoding latency. This choice then impacts the FEC-related latency of all receivers, even those experiencing a good communication quality, since no FEC encoding can happen until all the source data of the block is available at the sender, which directly depends on the block size.

1.2. Lower Latency and Better Protection of Real-Time Flows with the Sliding Window RLC Codes

This document introduces two fully specified FEC schemes that do not follow the block code approach: the Sliding Window Random Linear Codes (RLC) over either Galois Fields (a.k.a., Finite Fields) GF(2) (the "binary case") or GF(2^8), each time with the possibility of controlling the code density. These FEC schemes are used to protect arbitrary media streams along the lines defined by FECFRAME extended to Sliding Window FEC Codes [RFC8680]. These FEC schemes and, more generally, Sliding Window FEC Codes are recommended, for instance, with media that feature real-time constraints sent within a multicast/broadcast session [Roca17].

The RLC codes belong to the broad class of Sliding Window AL-FEC Codes (a.k.a., convolutional codes) [RFC8406]. The encoding process is based on an encoding window that slides over the set of source packets (in fact source symbols as we will see in Section 3.2), this window being either of fixed size or variable size (a.k.a., an elastic window). Repair symbols are generated on-the-fly, by computing a random linear combination of the source symbols present in the current encoding window, and passed to the transport layer.

At the receiver, a linear system is managed from the set of received source and repair packets. New variables (representing source symbols) and equations (representing the linear combination carried by each repair symbol received) are added upon receiving new packets. Variables and the equations they are involved in are removed when they are too old with respect to their validity period (real-time constraints). Lost source symbols are then recovered thanks to this linear system whenever its rank permits to solve it (at least partially).

The protection of any multicast/broadcast session needs to be dimensioned by considering the worst communication conditions one wants to support. This is also true with RLC (more generally, any sliding window) code. However, the receivers experiencing a good to medium communication quality will observe a reduced FEC-related latency compared to block codes [Roca17] since an isolated lost source packet is quickly recovered with the following repair packet. On the opposite, with a block code, recovering an isolated lost source packet always requires waiting for the first repair packet to arrive after the end of the block. Additionally, under certain situations (e.g., with a limited FEC-related latency budget and with constant bitrate transmissions after FECFRAME encoding), Sliding Window Codes can more efficiently achieve a target transmission quality (e.g., measured by the residual loss after FEC decoding) by sending fewer repair packets (i.e., higher code rate) than block codes.

1.3. Small Transmission Overheads with the Sliding Window RLC FEC Scheme

The Sliding Window RLC FEC scheme is designed to limit the packet header overhead. The main requirement is that each repair packet header must enable a receiver to reconstruct the set of source symbols plus the associated coefficients used during the encoding process. In order to minimize packet overhead, the set of source symbols in the encoding window as well as the set of

coefficients over GF(2^m) (where m is 1 or 8, depending on the FEC scheme) used in the linear combination are not individually listed in the repair packet header. Instead, each FEC Repair Packet header contains:

- the Encoding Symbol Identifier (ESI) of the first source symbol in the encoding window as well as the number of symbols (since this number may vary with a variable size, elastic window). These two pieces of information enable each receiver to reconstruct the set of source symbols considered during encoding, the only constraint being that there cannot be any gap;
- the seed and density threshold parameters used by a coding coefficients generation function (Section 3.6). These two pieces of information enable each receiver to generate the same set of coding coefficients over GF(2^m) as the sender;

Therefore, no matter the number of source symbols present in the encoding window, each FEC Repair Packet features a fixed 64-bit long header, called Repair FEC Payload ID (Figure 8). Similarly, each FEC Source Packet features a fixed 32-bit long trailer, called Explicit Source FEC Payload ID (Figure 6), that contains the ESI of the first source symbol (Section 3.2).

1.4. Document Organization

This fully-specified FEC scheme follows the structure required by [RFC6363], Section 5.6 ("FEC Scheme Requirements"), namely:

1. Procedures: This section describes procedures specific to this FEC scheme, namely: RLC parameters derivation, ADUI and source symbols mapping, pseudorandom number generator, and coding coefficients generation function;
2. Formats and Codes: This section defines the Source FEC Payload ID and Repair FEC Payload ID formats, carrying the signaling information associated to each source or repair symbol. It also defines the FEC Framework Configuration Information (FFCI) carrying signaling information for the session;
3. FEC Code Specification: Finally this section provides the code specification.

2. Definitions and Abbreviations

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.

This document uses the following definitions and abbreviations:

a^b a to the power of b

GF(q) denotes a finite field (also known as the Galois Field) with q elements. We assume that q = 2^m in this document

m defines the length of the elements in the finite field, in bits. In this document, m is equal to 1 or 8

ADU: Application Data Unit

ADUI: Application Data Unit Information (includes the F, L and padding fields in addition to the ADU)

E: size of an encoding symbol (i.e., source or repair symbol), assumed fixed (in bytes)

br_in: transmission bitrate at the input of the FECFRAME sender, assumed fixed (in bits/s)

br_out: transmission bitrate at the output of the FECFRAME sender, assumed fixed (in bits/s)

max_lat: maximum FEC-related latency within FECFRAME (a decimal number expressed in seconds)

cr: RLC coding rate, ratio between the total number of source symbols and the total number of source plus repair symbols

ew_size: encoding window current size at a sender (in symbols)

ew_max_size: encoding window maximum size at a sender (in symbols)

dw_max_size: decoding window maximum size at a receiver (in symbols)

ls_max_size: linear system maximum size (or width) at a receiver (in symbols)

WSR: window size ratio parameter used to derive ew_max_size (encoder) and ls_max_size (decoder).

PRNG: pseudorandom number generator

TinyMT32: PRNG used in this specification.

DT: coding coefficients density threshold, an integer between 0 and 15 (inclusive) the controls the fraction of coefficients that are nonzero

3. Common Procedures

This section introduces the procedures that are used by these FEC schemes.

3.1. Codec Parameters

A codec implementing the Sliding Window RLC FEC scheme relies on several parameters:

Maximum FEC-related latency budget, max_lat (a decimal number expressed in seconds) with real-time flows:

a source ADU flow can have real-time constraints, and therefore any FECFRAME related operation should take place within the validity period of each ADU (Appendix D describes an exception to this rule). When there are multiple flows with different real-time constraints, we consider the most stringent constraints (see item 6 in Section 10.2 of

[RFC6363], for recommendations when several flows are globally protected). The maximum FEC-related latency budget, max_lat, accounts for all sources of latency added by FEC encoding (at a sender) and FEC decoding (at a receiver). Other sources of latency (e.g., added by network communications) are out of scope and must be considered separately (said differently, they have already been deducted from max_lat). max_lat can be regarded as the latency budget permitted for all FEC-related operations. This is an input parameter that enables a FECFRAME sender to derive other internal parameters (see Appendix C);

Encoding window current (resp. maximum) size, ew_size (resp. ew_max_size) (in symbols):
: at a FECFRAME sender, during FEC encoding, a repair symbol is computed as a linear combination of the ew_size source symbols present in the encoding window. The ew_max_size is the maximum size of this window, while ew_size is the current size. For example, in the common case at session start, upon receiving new source ADUs, the ew_size progressively increases until it reaches its maximum value, ew_max_size. We have:

$$0 < ew_size <= ew_max_size$$

Decoding window maximum size, dw_max_size (in symbols):
: at a FECFRAME receiver, dw_max_size is the maximum number of received or lost source symbols that are still within their latency budget;

Linear system maximum size, ls_max_size (in symbols):
: at a FECFRAME receiver, the linear system maximum size, ls_max_size, is the maximum number of received or lost source symbols in the linear system (i.e., the variables). It **SHOULD NOT** be smaller than dw_max_size since it would mean that, even after receiving a sufficient number of FEC Repair Packets, a lost ADU may not be recovered just because the associated source symbols have been prematurely removed from the linear system, which is usually counter-productive. On the opposite, the linear system **MAY** grow beyond the dw_max_size (Appendix D);

Symbol size, E (in bytes):
: the E parameter determines the source and repair symbol sizes (necessarily equal). This is an input parameter that enables a FECFRAME sender to derive other internal parameters, as explained below. An implementation at a sender **MUST** fix the E parameter and **MUST** communicate it as part of the FEC Scheme-Specific Information (Section 4.1.1.2).

Code rate, cr:
: The code rate parameter determines the amount of redundancy added to the flow. More precisely the cr is the ratio between the total number of source symbols and the total number of source plus repair symbols and by definition: 0 < cr <= 1. This is an input parameter that enables a FECFRAME sender to derive other internal parameters, as explained below. However, there is no need to communicate the cr parameter per see (it's not required to process a repair symbol at a receiver). This code rate parameter can be static. However, in specific use-cases (e.g., with unicast transmissions in presence of a feedback mechanism that estimates the communication quality, out of scope of FECFRAME), the code rate may be adjusted dynamically.

Appendix C proposes non-normative techniques to derive those parameters, depending on the use-case specificities.

3.2. ADU, ADUI, and Source Symbols Mappings

At a sender, an ADU coming from the application is not directly mapped to source symbols. When multiple source flows (e.g., media streams) are mapped onto the same FECFRAME instance, each flow is assigned its own Flow ID value (see below). This Flow ID is then prepended to each ADU before FEC encoding. This way, FEC decoding at a receiver also recovers this Flow ID and the recovered ADU can be assigned to the right source flow (note that the 5-tuple used to identify the right source flow of a received ADU is absent with a recovered ADU since it is not FEC protected).

Additionally, since ADUs are of variable size, padding is needed so that each ADU (with its flow identifier) contribute to an integral number of source symbols. This requires adding the original ADU length to each ADU before doing FEC encoding. Because of these requirements, an intermediate format, the ADUI, or ADU Information, is considered [RFC6363].

For each incoming ADU, an ADUI **MUST** be created as follows. First of all, 3 bytes are prepended (Figure 1):

Flow ID (F) (8-bit field): this unsigned byte contains the integer identifier associated to the source ADU flow to which this ADU belongs. It is assumed that a single byte is sufficient, which implies that no more than 256 flows will be protected by a single FECFRAME session instance.

Length (L) (16-bit field): this unsigned integer contains the length of this ADU, in network byte order (i.e., big endian). This length is for the ADU itself and does not include the F, L, or Pad fields.

Then, zero padding is added to the ADU if needed:

Padding (Pad) (variable size field): this field contains zero padding to align the F, L, ADU and padding up to a size that is multiple of E bytes (i.e., the source and repair symbol length).

The data unit resulting from the ADU and the F, L, and Pad fields is called ADUI. Since ADUs can have different sizes, this is also the case for ADUIs. However, an ADUI always contributes to an integral number of source symbols.

```
   symbol length, E              E                     E
< ------------------ >< ------------------ >< ------------------ >
+-+--+-----------------------------------------------+-------------+
|F| L|                      ADU                      |     Pad     |
+-+--+-----------------------------------------------+-------------+
```

Figure 1: ADUI Creation Example, Resulting in Three Source Symbols

Note that neither the initial 3 bytes nor the optional padding are sent over the network. However, they are considered during FEC encoding, and a receiver that lost a certain FEC Source Packet (e.g., the UDP datagram containing this FEC Source Packet when UDP is used as the transport protocol) will be able to recover the ADUI if FEC decoding succeeds. Thanks to the initial 3 bytes, this receiver will get rid of the padding (if any) and identify the corresponding ADU flow.

3.3. Encoding Window Management

Source symbols and the corresponding ADUs are removed from the encoding window:

- when the sliding encoding window has reached its maximum size, ew_max_size. In that case the oldest symbol **MUST** be removed before adding a new symbol, so that the current encoding window size always remains inferior or equal to the maximum size: ew_size <= ew_max_size;
- when an ADU has reached its maximum validity duration in case of a real-time flow. When this happens, all source symbols corresponding to the ADUI that expired **SHOULD** be removed from the encoding window;

Source symbols are added to the sliding encoding window each time a new ADU arrives, once the ADU-to-source symbols mapping has been performed (Section 3.2). The current size of the encoding window, ew_size, is updated after adding new source symbols. This process may require to remove old source symbols so that: ew_size <= ew_max_size.

Note that a FEC codec may feature practical limits in the number of source symbols in the encoding window (e.g., for computational complexity reasons). This factor may further limit the ew_max_size value, in addition to the maximum FEC-related latency budget (Section 3.1).

3.4. Source Symbol Identification

Each source symbol is identified by an Encoding Symbol ID (ESI), an unsigned integer. The ESI of source symbols **MUST** start with value 0 for the first source symbol and **MUST** be managed sequentially. Wrapping to zero happens after reaching the maximum value made possible by the ESI field size (this maximum value is FEC scheme dependent, for instance, 2^{32}-1 with FEC schemes 9 and 10).

No such consideration applies to repair symbols.

3.5. Pseudorandom Number Generator (PRNG)

In order to compute coding coefficients (see Section 3.6), the RLC FEC schemes rely on the TinyMT32 PRNG defined in [RFC8682] with two additional functions defined in this section.

This PRNG **MUST** first be initialized with a 32-bit unsigned integer, used as a seed, with:

```
void   tinymt32_init (tinymt32_t * s, uint32_t seed);
```

With the FEC schemes defined in this document, the seed is in practice restricted to a value between 0 and 0xFFFF inclusive (note that this PRNG accepts a seed value equal to 0), since this is the Repair_Key 16-bit field value of the Repair FEC Payload ID (Section 4.1.3). In practice, how to manage the seed and Repair_Key values (both are equal) is left to the implementer, using a monotonically increasing counter being one possibility (Section 6.1). In addition to the seed, this function takes as parameter a pointer to an instance of a tinymt32_t structure that is used to keep the internal state of the PRNG.

Then, each time a new pseudorandom integer between 0 and 15 inclusive (4-bit pseudorandom integer) is needed, the following function is used:

```
uint32_t   tinymt32_rand16 (tinymt32_t * s);
```

This function takes as parameter a pointer to the same tinymt32_t structure (that is left unchanged between successive calls to the function).

Similarly, each time a new pseudorandom integer between 0 and 255 inclusive (8-bit pseudorandom integer) is needed, the following function is used:

```
uint32_t   tinymt32_rand256 (tinymt32_t * s);
```

These two functions keep respectively the 4 or 8 less significant bits of the 32-bit pseudorandom number generated by the tinymt32_generate_uint32() function of [RFC8682]. This is done by computing the result of a binary AND between the tinymt32_generate_uint32() output and respectively the 0xF or 0xFF constants, using 32-bit unsigned integer operations. Figure 2 shows a possible implementation. This is a C language implementation, written for C99 [C99]. Test results discussed in Appendix B show that this simple technique, applied to this PRNG, is in line with the RLC FEC schemes needs.

```
<CODE BEGINS>
/**
 * This function outputs a pseudorandom integer in [0 .. 15] range.
 *
 * @param s     pointer to tinymt internal state.
 * @return      unsigned integer between 0 and 15 inclusive.
 */
uint32_t tinymt32_rand16(tinymt32_t *s)
{
    return (tinymt32_generate_uint32(s) & 0xF);
}

/**
 * This function outputs a pseudorandom integer in [0 .. 255] range.
 *
 * @param s     pointer to tinymt internal state.
 * @return      unsigned integer between 0 and 255 inclusive.
 */
uint32_t tinymt32_rand256(tinymt32_t *s)
{
    return (tinymt32_generate_uint32(s) & 0xFF);
}

<CODE ENDS>
```

Figure 2: 4-bit and 8-bit Mapping Functions for TinyMT32

Any implementation of this PRNG **MUST** have the same output as that provided by the reference implementation of [RFC8682]. In order to increase the compliance confidence, three criteria are proposed: the one described in [RFC8682] (for the TinyMT32 32-bit unsigned integer generator), and the two others detailed in Appendix A (for the mapping to 4-bit and 8-bit intervals). Because of the way the mapping functions work, it is unlikely that an implementation that fulfills the first criterion fails to fulfill the two others.

3.6. Coding Coefficients Generation Function

The coding coefficients used during the encoding process are generated at the RLC encoder by the generate_coding_coefficients() function each time a new repair symbol needs to be produced. The fraction of coefficients that are nonzero (i.e., the density) is controlled by the DT (Density Threshold) parameter. DT has values between 0 (the minimum value) and 15 (the maximum value), and the average probability of having a nonzero coefficient equals (DT + 1) / 16. In particular, when DT equals 15 the function guaranties that all coefficients are nonzero (i.e., maximum density).

These considerations apply to both the RLC over GF(2) and RLC over GF(2^8), the only difference being the value of the m parameter. With the RLC over GF(2) FEC scheme (Section 5), m is equal to 1. With RLC over GF(2^8) FEC scheme (Section 4), m is equal to 8.

Figure 3 shows the reference generate_coding_coefficients() implementation. This is a C language implementation, written for C99 [C99].

```
<CODE BEGINS>
#include <string.h>

/*
 * Fills in the table of coding coefficients (of the right size)
 * provided with the appropriate number of coding coefficients to
 * use for the repair symbol key provided.
 *
 * (in) repair_key    key associated to this repair symbol. This
 *                    parameter is ignored (useless) if m=1 and dt=15
 * (in/out) cc_tab    pointer to a table of the right size to store
 *                    coding coefficients. All coefficients are
 *                    stored as bytes, regardless of the m parameter,
 *                    upon return of this function.
 * (in) cc_nb         number of entries in the cc_tab table. This
 *                    value is equal to the current encoding window
 *                    size.
 * (in) dt            integer between 0 and 15 (inclusive) that
 *                    controls the density. With value 15, all
 *                    coefficients are guaranteed to be nonzero
 *                    (i.e., equal to 1 with GF(2) and equal to a
 *                    value in {1,... 255} with GF(2^^8)), otherwise
 *                    a fraction of them will be 0.
 * (in) m             Finite Field GF(2^^m) parameter. In this
 *                    document only values 1 and 8 are considered.
 * (out)              returns 0 in case of success, an error code
 *                    different than 0 otherwise.
 */
int generate_coding_coefficients (uint16_t  repair_key,
                                  uint8_t*  cc_tab,
                                  uint16_t  cc_nb,
                                  uint8_t   dt,
                                  uint8_t   m)
{
    uint32_t      i;
    tinymt32_t    s;    /* PRNG internal state */

    if (dt > 15) {
        return -1; /* error, bad dt parameter */
    }
    switch (m) {
    case 1:
        if (dt == 15) {
            /* all coefficients are 1 */
            memset(cc_tab, 1, cc_nb);
        } else {
            /* here coefficients are either 0 or 1 */
            tinymt32_init(&s, repair_key);
            for (i = 0 ; i < cc_nb ; i++) {
                cc_tab[i] = (tinymt32_rand16(&s) <= dt) ? 1 : 0;
            }
        }
        break;

    case 8:
        tinymt32_init(&s, repair_key);
        if (dt == 15) {
```

```
            /* coefficient 0 is avoided here in order to include
             * all the source symbols */
            for (i = 0 ; i < cc_nb ; i++) {
                do {
                    cc_tab[i] = (uint8_t) tinymt32_rand256(&s);
                } while (cc_tab[i] == 0);
            }
        } else {
            /* here a certain number of coefficients should be 0 */
            for (i = 0 ; i < cc_nb ; i++) {
                if (tinymt32_rand16(&s) <= dt) {
                    do {
                        cc_tab[i] = (uint8_t) tinymt32_rand256(&s);
                    } while (cc_tab[i] == 0);
                } else {
                    cc_tab[i] = 0;
                }
            }
        }
        break;

    default:
        return -2; /* error, bad parameter m */
    }
    return 0; /* success */
}

<CODE ENDS>
```

Figure 3: Reference Implementation of the Coding Coefficients Generation Function

3.7. Finite Field Operations

3.7.1. Finite Field Definitions

The two RLC FEC schemes specified in this document reuse the Finite Fields defined in [RFC5510], Section 8.1. More specifically, the elements of the field GF(2^m) are represented by polynomials with binary coefficients (i.e., over GF(2)) and degree lower or equal to m-1. The addition between two elements is defined as the addition of binary polynomials in GF(2), which is equivalent to a bitwise XOR operation on the binary representation of these elements.

With GF(2^8), multiplication between two elements is the multiplication modulo a given irreducible polynomial of degree 8. The following irreducible polynomial is used for GF(2^8):

$$x^8 + x^4 + x^3 + x^2 + 1$$

With GF(2), multiplication corresponds to a logical AND operation.

3.7.2. Linear Combination of Source Symbol Computation

The two RLC FEC schemes require the computation of a linear combination of source symbols, using the coding coefficients produced by the generate_coding_coefficients() function and stored in the cc_tab[] array.

With the RLC over GF(2^8) FEC scheme, a linear combination of the ew_size source symbol present in the encoding window, say src_0 to src_ew_size_1, in order to generate a repair symbol, is computed as follows. For each byte of position i in each source and the repair symbol, where i belongs to [0; E-1], compute:

```
repair[i] = cc_tab[0] * src_0[i] XOR cc_tab[1] * src_1[i] XOR ...
XOR cc_tab[ew_size - 1] * src_ew_size_1[i]
```

where * is the multiplication over GF(2^8). In practice various optimizations need to be used in order to make this computation efficient (see in particular [PGM13]).

With the RLC over GF(2) FEC scheme (binary case), a linear combination is computed as follows. The repair symbol is the XOR sum of all the source symbols corresponding to a coding coefficient cc_tab[j] equal to 1 (i.e., the source symbols corresponding to zero coding coefficients are ignored). The XOR sum of the byte of position i in each source is computed and stored in the corresponding byte of the repair symbol, where i belongs to [0; E-1]. In practice, the XOR sums will be computed several bytes at a time (e.g., on 64 bit words, or on arrays of 16 or more bytes when using SIMD CPU extensions).

With both FEC schemes, the details of how to optimize the computation of these linear combinations are of high practical importance but out of scope of this document.

4. Sliding Window RLC FEC Scheme over GF(2^8) for Arbitrary Packet Flows

This fully-specified FEC scheme defines the Sliding Window Random Linear Codes (RLC) over GF (2^8).

4.1. Formats and Codes

4.1.1. FEC Framework Configuration Information

Following the guidelines of Section 5.6 of [RFC6363], this section provides the FEC Framework Configuration Information (or FFCI). This FCCI needs to be shared (e.g., using SDP) between the FECFRAME sender and receiver instances in order to synchronize them. It includes a FEC Encoding ID, mandatory for any FEC scheme specification, plus scheme-specific elements.

4.1.1.1. FEC Encoding ID

FEC Encoding ID: the value assigned to this fully specified FEC scheme **MUST** be 10, as assigned by IANA (Section 9).

When SDP is used to communicate the FFCI, this FEC Encoding ID is carried in the 'encoding-id' parameter.

4.1.1.2. FEC Scheme-Specific Information

The FEC Scheme-Specific Information (FSSI) includes elements that are specific to the present FEC scheme. More precisely:

Encoding symbol size (E): a non-negative integer that indicates the size of each encoding symbol in bytes;

Window Size Ratio (WSR) parameter: a non-negative integer between 0 and 255 (both inclusive) used to initialize window sizes. A value of 0 indicates this parameter is not considered (e.g., a fixed encoding window size may be chosen). A value between 1 and 255 inclusive is required by certain of the parameter derivation techniques described in Appendix C;

This element is required both by the sender (RLC encoder) and the receiver(s) (RLC decoder).

When SDP is used to communicate the FFCI, this FEC Scheme-Specific Information is carried in the 'fssi' parameter in textual representation as specified in [RFC6364]. For instance:

```
fssi=E:1400,WSR:191
```

In that case the name values "E" and "WSR" are used to convey the E and WSR parameters respectively.

If another mechanism requires the FSSI to be carried as an opaque octet string, the encoding format consists of the following three octets, where the E field is carried in "big-endian" or "network order" format, that is, most significant byte first:

Encoding symbol length (E): 16-bit field;

Window Size Ratio Parameter (WSR): 8-bit field.

These three octets can be communicated as such, or for instance, be subject to an additional Base64 encoding.

```
 0                   1                   2
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Encoding Symbol Length (E)  |      WSR      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4: FSSI Encoding Format

4.1.2. Explicit Source FEC Payload ID

A FEC Source Packet **MUST** contain an Explicit Source FEC Payload ID that is appended to the end of the packet as illustrated in Figure 5.

```
+--------------------------------+
|           IP Header            |
+--------------------------------+
|        Transport Header        |
+--------------------------------+
|              ADU               |
+--------------------------------+
| Explicit Source FEC Payload ID |
+--------------------------------+
```

Figure 5: Structure of an FEC Source Packet with the Explicit Source FEC Payload ID

More precisely, the Explicit Source FEC Payload ID is composed of the following field, carried in "big-endian" or "network order" format, that is, most significant byte first (Figure 6):

Encoding Symbol ID (ESI) (32-bit field): this unsigned integer identifies the first source symbol of the ADUI corresponding to this FEC Source Packet. The ESI is incremented for each new source symbol, and after reaching the maximum value (2^{32}-1), wrapping to zero occurs.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Encoding Symbol ID (ESI)                   |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 6: Source FEC Payload ID Encoding Format

4.1.3. Repair FEC Payload ID

A FEC Repair Packet **MAY** contain one or more repair symbols. When there are several repair symbols, all of them **MUST** have been generated from the same encoding window, using Repair_Key values that are managed as explained below. A receiver can easily deduce the number of repair symbols within a FEC Repair Packet by comparing the received FEC Repair Packet size (equal to the UDP payload size when UDP is the underlying transport protocol) and the symbol size, E, communicated in the FFCI.

A FEC Repair Packet **MUST** contain a Repair FEC Payload ID that is prepended to the repair symbol as illustrated in Figure 7.

```
+--------------------------------+
|           IP Header            |
+--------------------------------+
|        Transport Header        |
+--------------------------------+
|     Repair FEC Payload ID      |
+--------------------------------+
|         Repair Symbol          |
+--------------------------------+
```

Figure 7: Structure of an FEC Repair Packet with the Repair FEC Payload ID

More precisely, the Repair FEC Payload ID is composed of the following fields where all integer fields are carried in "big-endian" or "network order" format, that is, most significant byte first (Figure 8):

Repair_Key (16-bit field): this unsigned integer is used as a seed by the coefficient generation function (Section 3.6) in order to generate the desired number of coding coefficients. This repair key may be a monotonically increasing integer value that loops back to 0 after reaching 65535 (see Section 6.1). When a FEC Repair Packet contains several repair symbols, this repair key value is that of the first repair symbol. The remaining repair keys can be deduced by incrementing by 1 this value, up to a maximum value of 65535 after which it loops back to 0.

Density Threshold for the coding coefficients, DT (4-bit field): this unsigned integer carries the Density Threshold (DT) used by the coding coefficient generation function Section 3.6. More precisely, it controls the probability of having a nonzero coding coefficient, which equals (DT+1) / 16. When a FEC Repair Packet contains several repair symbols, the DT value applies to all of them;

Number of Source Symbols in the encoding window, NSS (12-bit field): this unsigned integer indicates the number of source symbols in the encoding window when this repair symbol was generated. When a FEC Repair Packet contains several repair symbols, this NSS value applies to all of them;

ESI of First Source Symbol in the encoding window, FSS_ESI (32-bit field): this unsigned integer indicates the ESI of the first source symbol in the encoding window when this repair symbol was generated. When a FEC Repair Packet contains several repair symbols, this FSS_ESI value applies to all of them;

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|       Repair_Key              |  DT   |NSS (# src symb in ew) |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            FSS_ESI                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 8: Repair FEC Payload ID Encoding Format

4.2. Procedures

All the procedures of Section 3 apply to this FEC scheme.

5. Sliding Window RLC FEC Scheme over GF(2) for Arbitrary Packet Flows

This fully-specified FEC scheme defines the Sliding Window Random Linear Codes (RLC) over GF (2) (binary case).

5.1. Formats and Codes

5.1.1. FEC Framework Configuration Information

5.1.1.1. FEC Encoding ID

FEC Encoding ID: the value assigned to this fully specified FEC scheme **MUST** be 9, as assigned by IANA (Section 9).

When SDP is used to communicate the FFCI, this FEC Encoding ID is carried in the 'encoding-id' parameter.

5.1.1.2. FEC Scheme-Specific Information

All the considerations of Section 4.1.1.2 apply here.

5.1.2. Explicit Source FEC Payload ID

All the considerations of Section 4.1.2 apply here.

5.1.3. Repair FEC Payload ID

All the considerations of Section 4.1.3 apply here, with the only exception that the Repair_Key field is useless if DT = 15 (indeed, in that case all the coefficients are necessarily equal to 1 and the coefficient generation function does not use any PRNG). When DT = 15 the FECFRAME sender **MUST** set the Repair_Key field to zero on transmission and a receiver **MUST** ignore it on receipt.

5.2. Procedures

All the procedures of Section 3 apply to this FEC scheme.

6. FEC Code Specification

6.1. Encoding Side

This section provides a high level description of a Sliding Window RLC encoder.

Whenever a new FEC Repair Packet is needed, the RLC encoder instance first gathers the ew_size source symbols currently in the sliding encoding window. Then it chooses a repair key, which can be a monotonically increasing integer value, incremented for each repair symbol up to a maximum value of 65535 (as it is carried within a 16-bit field) after which it loops back to 0. This repair key is communicated to the coefficient generation function (Section 3.6) in order to generate ew_size coding coefficients. Finally, the FECFRAME sender computes the repair symbol as a linear combination of the ew_size source symbols using the ew_size coding coefficients (Section 3.7). When E is small and when there is an incentive to pack several repair symbols within the same FEC Repair Packet, the appropriate number of repair symbols are computed. In that case the repair key for each of them **MUST** be incremented by 1, keeping the same ew_size source symbols, since only the first repair key will be carried in the Repair FEC Payload ID. The FEC Repair Packet can then be passed to the transport layer for transmission. The source versus repair FEC packet transmission order is out of scope of this document and several approaches exist that are implementation-specific.

Other solutions are possible to select a repair key value when a new FEC Repair Packet is needed, for instance, by choosing a random integer between 0 and 65535. However, selecting the same repair key as before (which may happen in case of a random process) is only meaningful if the encoding window has changed, otherwise the same FEC Repair Packet will be generated. In any case, choosing the repair key is entirely at the discretion of the sender, since it is communicated to the receiver(s) in each Repair FEC Payload ID. A receiver should not make any assumption on the way the repair key is managed.

6.2. Decoding Side

This section provides a high level description of a Sliding Window RLC decoder.

A FECFRAME receiver needs to maintain a linear system whose variables are the received and lost source symbols. Upon receiving a FEC Repair Packet, a receiver first extracts all the repair symbols it contains (in case several repair symbols are packed together). For each repair symbol, when at least one of the corresponding source symbols it protects has been lost, the receiver adds an equation to the linear system (or no equation if this repair packet does not change the linear system rank). This equation of course re-uses the ew_size coding coefficients that are computed by the same coefficient generation function (Section 3.6), using the repair key and encoding window descriptions carried in the Repair FEC Payload ID. Whenever possible (i.e., when a sub-system covering one or more lost source symbols is of full rank), decoding is performed in order to recover lost source symbols. Gaussian elimination is one possible algorithm to solve this linear

system. Each time an ADUI can be totally recovered, padding is removed (thanks to the Length field, L, of the ADUI) and the ADU is assigned to the corresponding application flow (thanks to the Flow ID field, F, of the ADUI). This ADU is finally passed to the corresponding upper application. Received FEC Source Packets, containing an ADU, **MAY** be passed to the application either immediately or after some time to guaranty an ordered delivery to the application. This document does not mandate any approach as this is an operational and management decision.

With real-time flows, a lost ADU that is decoded after the maximum latency or an ADU received after this delay has no value to the application. This raises the question of deciding whether or not an ADU is late. This decision **MAY** be taken within the FECFRAME receiver (e.g., using the decoding window, see Section 3.1) or within the application (e.g., using RTP timestamps within the ADU). Deciding which option to follow and whether or not to pass all ADUs, including those assumed late, to the application are operational decisions that depend on the application and are therefore out of scope of this document. Additionally, Appendix D discusses a backward compatible optimization whereby late source symbols **MAY** still be used within the FECFRAME receiver in order to improve transmission robustness.

7. Security Considerations

The FEC Framework document [RFC6363] provides a fairly comprehensive analysis of security considerations applicable to FEC schemes. Therefore, the present section follows the security considerations section of [RFC6363] and only discusses specific topics.

7.1. Attacks Against the Data Flow

7.1.1. Access to Confidential Content

The Sliding Window RLC FEC scheme specified in this document does not change the recommendations of [RFC6363]. To summarize, if confidentiality is a concern, it is **RECOMMENDED** that one of the solutions mentioned in [RFC6363] is used with special considerations to the way this solution is applied (e.g., is encryption applied before or after FEC protection, within the end system or in a middlebox), to the operational constraints (e.g., performing FEC decoding in a protected environment may be complicated or even impossible) and to the threat model.

7.1.2. Content Corruption

The Sliding Window RLC FEC scheme specified in this document does not change the recommendations of [RFC6363]. To summarize, it is **RECOMMENDED** that one of the solutions mentioned in [RFC6363] is used on both the FEC Source and Repair Packets.

7.2. Attacks Against the FEC Parameters

The FEC scheme specified in this document defines parameters that can be the basis of attacks. More specifically, the following parameters of the FFCI may be modified by an attacker who targets receivers (Section 4.1.1.2):

FEC Encoding ID:

changing this parameter leads a receiver to consider a different FEC scheme. The consequences are severe, the format of the Explicit Source FEC Payload ID and Repair FEC Payload ID of received packets will probably differ, leading to various malfunctions. Even if the original and modified FEC schemes share the same format, FEC decoding will either fail or lead to corrupted decoded symbols. This will happen if an attacker turns value 9 (i.e., RLC over GF(2)) to value 10 (RLC over GF(2^8)), an additional consequence being a higher processing overhead at the receiver. In any case, the attack results in a form of Denial of Service (DoS) or corrupted content.

Encoding symbol length (E): setting this E parameter to a different value will confuse a receiver. If the size of a received FEC Repair Packet is no longer multiple of the modified E value, a receiver quickly detects a problem and **SHOULD** reject the packet. If the new E value is a sub-multiple of the original E value (e.g., half the original value), then receivers may not detect the problem immediately. For instance, a receiver may think that a received FEC Repair Packet contains more repair symbols (e.g., twice as many if E is reduced by half), leading to malfunctions whose nature depends on implementation details. Here also, the attack always results in a form of DoS or corrupted content.

It is therefore **RECOMMENDED** that security measures be taken to guarantee the FFCI integrity, as specified in [RFC6363]. How to achieve this depends on the way the FFCI is communicated from the sender to the receiver, which is not specified in this document.

Similarly, attacks are possible against the Explicit Source FEC Payload ID and Repair FEC Payload ID. More specifically, in case of a FEC Source Packet, the following value can be modified by an attacker who targets receivers:

Encoding Symbol ID (ESI): changing the ESI leads a receiver to consider a wrong ADU, resulting in severe consequences, including corrupted content passed to the receiving application;

And in case of a FEC Repair Packet:

Repair Key: changing this value leads a receiver to generate a wrong coding coefficient sequence, and therefore any source symbol decoded using the repair symbols contained in this packet will be corrupted;

DT: changing this value also leads a receiver to generate a wrong coding coefficient sequence, and therefore any source symbol decoded using the repair symbols contained in this packet will be corrupted. In addition, if the DT value is significantly increased, it will generate a higher processing overhead at a receiver. In case of very large encoding windows, this may impact the terminal performance;

NSS: changing this value leads a receiver to consider a different set of source symbols, and therefore any source symbol decoded using the repair symbols contained in this packet will be corrupted. In addition, if the NSS value is significantly increased, it will generate a higher processing overhead at a receiver, which may impact the terminal performance;

FSS_ESI:

changing this value also leads a receiver to consider a different set of source symbols and therefore any source symbol decoded using the repair symbols contained in this packet will be corrupted.

It is therefore **RECOMMENDED** that security measures are taken to guarantee the FEC Source and Repair Packets as stated in [RFC6363].

7.3. When Several Source Flows are to be Protected Together

The Sliding Window RLC FEC scheme specified in this document does not change the recommendations of [RFC6363].

7.4. Baseline Secure FEC Framework Operation

The Sliding Window RLC FEC scheme specified in this document does not change the recommendations of [RFC6363] concerning the use of the IPsec/Encapsulating Security Payload (ESP) security protocol as a mandatory-to-implement (but not mandatory-to-use) security scheme. This is well suited to situations where the only insecure domain is the one over which the FEC Framework operates.

7.5. Additional Security Considerations for Numerical Computations

In addition to the above security considerations, inherited from [RFC6363], the present document introduces several formulae, in particular in Appendix C.1. It is **RECOMMENDED** to check that the computed values stay within reasonable bounds since numerical overflows, caused by an erroneous implementation or an erroneous input value, may lead to hazardous behaviors. However, what "reasonable bounds" means is use-case and implementation dependent and is not detailed in this document.

Appendix C.2 also mentions the possibility of "using the timestamp field of an RTP packet header" when applicable. A malicious attacker may deliberately corrupt this header field in order to trigger hazardous behaviors at a FECFRAME receiver. Protection against this type of content corruption can be addressed with the above recommendations on a baseline secure operation. In addition, it is also **RECOMMENDED** to check that the timestamp value be within reasonable bounds.

8. Operations and Management Considerations

The FEC Framework document [RFC6363] provides a fairly comprehensive analysis of operations and management considerations applicable to FEC schemes. Therefore, the present section only discusses specific topics.

8.1. Operational Recommendations: Finite Field GF(2) Versus GF(2^8)

The present document specifies two FEC schemes that differ on the Finite Field used for the coding coefficients. It is expected that the RLC over GF(2^8) FEC scheme will be mostly used since it warrants a higher packet loss protection. In case of small encoding windows, the associated processing overhead is not an issue (e.g., we measured decoding speeds between 745 Mbps and 2.8 Gbps on an ARM Cortex-A15 embedded board in [Roca17] depending on the code rate and the channel conditions, using an encoding window of size 18 or 23 symbols; see the above article for the details). Of course the CPU overhead will increase with the encoding window size, because more operations in the GF(2^8) finite field will be needed.

The RLC over GF(2) FEC scheme offers an alternative. In that case operations symbols can be directly XOR-ed together which warrants high bitrate encoding and decoding operations, and can be an advantage with large encoding windows. However, packet loss protection is significantly reduced by using this FEC scheme.

8.2. Operational Recommendations: Coding Coefficients Density Threshold

In addition to the choice of the Finite Field, the two FEC schemes define a coding coefficient density threshold (DT) parameter. This parameter enables a sender to control the code density, i.e., the proportion of coefficients that are nonzero on average. With RLC over GF(2^8), it is usually appropriate that small encoding windows be associated to a density threshold equal to 15, the maximum value, in order to warrant a high loss protection.

On the opposite, with larger encoding windows, it is usually appropriate that the density threshold be reduced. With large encoding windows, an alternative can be to use RLC over GF(2) and a density threshold equal to 7 (i.e., an average density equal to 1/2) or smaller.

Note that using a density threshold equal to 15 with RLC over GF(2) is equivalent to using an XOR code that computes the XOR sum of all the source symbols in the encoding window. In that case: (1) only a single repair symbol can be produced for any encoding window, and (2) the repair_key parameter becomes useless (the coding coefficients generation function does not rely on the PRNG).

9. IANA Considerations

This document registers two values in the "FEC Framework (FECFRAME) FEC Encoding IDs" registry [RFC6363] as follows:

- 9 refers to the Sliding Window Random Linear Codes (RLC) over GF(2) FEC Scheme for Arbitrary Packet Flows, as defined in Section 5 of this document.
- 10 refers to the Sliding Window Random Linear Codes (RLC) over GF(2^8) FEC Scheme for Arbitrary Packet Flows, as defined in Section 4 of this document.

10. References

10.1. Normative References

[C99] International Organization for Standardization, "Programming languages - C: C99, correction 3:2007", ISO/IEC 9899:1999/Cor 3:2007, November 2007.

[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/info/rfc2119>.

[RFC6363] Watson, M., Begen, A., and V. Roca, "Forward Error Correction (FEC) Framework", RFC 6363, DOI 10.17487/RFC6363, October 2011, <https://www.rfc-editor.org/info/rfc6363>.

[RFC6364] Begen, A., "Session Description Protocol Elements for the Forward Error Correction (FEC) Framework", RFC 6364, DOI 10.17487/RFC6364, October 2011, <https://www.rfc-editor.org/info/rfc6364>.

[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/info/rfc8174>.

[RFC8680] Roca, V. and A. Begen, "Forward Error Correction (FEC) Framework Extension to Sliding Window Codes", RFC 8680, DOI 10.17487/RFC8680, January 2020, <https://www.rfc-editor.org/info/rfc8680>.

[RFC8682] Saito, M., Matsumoto, M., Roca, V., Ed., and E. Baccelli, "TinyMT32 Pseudorandom Number Generator (PRNG)", RFC 8682, DOI 10.17487/RFC8682, January 2020, <https://www.rfc-editor.org/info/rfc8682>.

10.2. Informative References

[PGM13] Plank, J., Greenan, K., and E. Miller, "A Complete Treatment of Software Implementations of Finite Field Arithmetic for Erasure Coding Applications", University of Tennessee Technical Report UT-CS-13-717, October 2013, <http://web.eecs.utk.edu/~plank/plank/papers/UT-CS-13-717.html>.

[RFC5170] Roca, V., Neumann, C., and D. Furodet, "Low Density Parity Check (LDPC) Staircase and Triangle Forward Error Correction (FEC) Schemes", RFC 5170, DOI 10.17487/RFC5170, June 2008, <https://www.rfc-editor.org/info/rfc5170>.

[RFC5510] Lacan, J., Roca, V., Peltotalo, J., and S. Peltotalo, "Reed-Solomon Forward Error Correction (FEC) Schemes", RFC 5510, DOI 10.17487/RFC5510, April 2009, <https://www.rfc-editor.org/info/rfc5510>.

[RFC6681] Watson, M., Stockhammer, T., and M. Luby, "Raptor Forward Error Correction (FEC) Schemes for FECFRAME", RFC 6681, DOI 10.17487/RFC6681, August 2012, <https://www.rfc-editor.org/info/rfc6681>.

[RFC6726] Paila, T., Walsh, R., Luby, M., Roca, V., and R. Lehtonen, "FLUTE - File Delivery over Unidirectional Transport", RFC 6726, DOI 10.17487/RFC6726, November 2012, <https://www.rfc-editor.org/info/rfc6726>.

[RFC6816] Roca, V., Cunche, M., and J. Lacan, "Simple Low-Density Parity Check (LDPC) Staircase Forward Error Correction (FEC) Scheme for FECFRAME", RFC 6816, DOI 10.17487/RFC6816, December 2012, <https://www.rfc-editor.org/info/rfc6816>.

[RFC6865] Roca, V., Cunche, M., Lacan, J., Bouabdallah, A., and K. Matsuzono, "Simple Reed-Solomon Forward Error Correction (FEC) Scheme for FECFRAME", RFC 6865, DOI 10.17487/RFC6865, February 2013, <https://www.rfc-editor.org/info/rfc6865>.

[RFC8406] Adamson, B., Adjih, C., Bilbao, J., Firoiu, V., Fitzek, F., Ghanem, S., Lochin, E., Masucci, A., Montpetit, M-J., Pedersen, M., Peralta, G., Roca, V., Ed., Saxena, P., and S. Sivakumar, "Taxonomy of Coding Techniques for Efficient Network Communications", RFC 8406, DOI 10.17487/RFC8406, June 2018, <https://www.rfc-editor.org/info/rfc8406>.

[Roca16] Roca, V., Teibi, B., Burdinat, C., Tran-Thai, T., and C. Thienot, "Block or Convolutional AL-FEC Codes? A Performance Comparison for Robust Low-Latency Communications", HAL ID hal-01395937v2, February 2017, <https://hal.inria.fr/hal-01395937/en/>.

[Roca17] Roca, V., Teibi, B., Burdinat, C., Tran, T., and C. Thienot, "Less Latency and Better Protection with AL-FEC Sliding Window Codes: a Robust Multimedia CBR Broadcast Case Study", 13th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob17), HAL ID hal-01571609, October 2017, <https://hal.inria.fr/hal-01571609v1/en/>.

Appendix A. TinyMT32 Validation Criteria (Normative)

PRNG determinism, for a given seed, is a requirement. Consequently, in order to validate an implementation of the TinyMT32 PRNG, the following criteria **MUST** be met.

The first criterion focuses on the tinymt32_rand256(), where the 32-bit integer of the core TinyMT32 PRNG is scaled down to an 8-bit integer. Using a seed value of 1, the first 50 values returned by: tinymt32_rand256() as 8-bit unsigned integers **MUST** be equal to values provided in Figure 9, to be read line by line.

```
        37        225        177        176         21
       246         54        139        168        237
       211        187         62        190        104
       135        210         99        176         11
       207         35         40        113        179
       214        254        101        212        211
       226         41        234        232        203
        29        194        211        112        107
       217        104        197        135         23
        89        210        252        109        166
```

Figure 9: First 50 decimal values (to be read per line) returned by tinymt32_rand256() as 8-bit unsigned integers, with a seed value of 1

The second criterion focuses on the tinymt32_rand16(), where the 32-bit integer of the core TinyMT32 PRNG is scaled down to a 4-bit integer. Using a seed value of 1, the first 50 values returned by: tinymt32_rand16() as 4-bit unsigned integers **MUST** be equal to values provided in Figure 10, to be read line by line.

```
         5          1          1          0          5
         6          6         11          8         13
         3         11         14         14          8
         7          2          3          0         11
        15          3          8          1          3
         6         14          5          4          3
         2          9         10          8         11
        13          2          3          0         11
         9          8          5          7          7
         9          2         12         13          6
```

Figure 10: First 50 decimal values (to be read per line) returned by tinymt32_rand16() as 4-bit unsigned integers, with a seed value of 1

Appendix B. Assessing the PRNG Adequacy (Informational)

This annex discusses the adequacy of the TinyMT32 PRNG and the tinymt32_rand16() and tinymt32_rand256() functions, to the RLC FEC schemes. The goal is to assess the adequacy of these two functions in producing coding coefficients that are sufficiently different from one another, across various repair symbols with repair key values in sequence (we can expect this approach to be commonly used by implementers, see Section 6.1). This section is purely informational and does not claim to be a solid evaluation.

The two RLC FEC schemes use the PRNG to produce pseudorandom coding coefficients (Section 3.6), each time a new repair symbol is needed. A different repair key is used for each repair symbol, usually by incrementing the repair key value (Section 6.1). For each repair symbol, a limited number of pseudorandom numbers is needed, depending on the DT and encoding window size (Section 3.6), using either tinymt32_rand16() or tinymt32_rand256(). Therefore, we

are more interested in the randomness of small sequences of random numbers mapped to 4-bit or 8-bit integers, than in the randomness of a very large sequence of random numbers which is not representative of the usage of the PRNG.

Evaluation of tinymt32_rand16(): We first generate a huge number (1,000,000,000) of small sequences (20 pseudorandom numbers per sequence), increasing the seed value for each sequence, and perform statistics on the number of occurrences of each of the 16 possible values across all sequences. In this first test we consider 32-bit seed values in order to assess the PRNG quality after output truncation to 4 bits.

Value	Occurrences	Percentage (%)
0	1250036799	6.2502
1	1249995831	6.2500
2	1250038674	6.2502
3	1250000881	6.2500
4	1250023929	6.2501
5	1249986320	6.2499
6	1249995587	6.2500
7	1250020363	6.2501
8	1249995276	6.2500
9	1249982856	6.2499
10	1249984111	6.2499
11	1250009551	6.2500
12	1249955768	6.2498
13	1249994654	6.2500
14	1250000569	6.2500
15	1249978831	6.2499

Table 1: tinymt32_rand16() Occurrence Statistics

Evaluation of tinymt32_rand16(): We first generate a huge number (1,000,000,000) of small sequences (20 pseudorandom numbers per sequence), increasing the seed value for each sequence, and perform statistics on the number of occurrences of each of the 16 possible values across the 20,000,000,000 numbers of all sequences. In this first test, we consider 32-bit seed values in order to assess the PRNG quality after output truncation to 4 bits.

The results (Table 1) show that all possible values are almost equally represented, or said differently, that the tinymt32_rand16() output converges to a uniform distribution where each of the 16 possible values would appear exactly 1 / 16 * 100 = 6.25% of times.

Since the RLC FEC schemes use of this PRNG will be limited to 16-bit seed values, we carried out the same test for the first 2^{16} seed values only. The distribution (not shown) is of course less uniform, with value occurrences ranging between 6.2121% (i.e., 81,423 occurrences out of a total of 65536*20=1,310,720) and 6.2948% (i.e., 82,507 occurrences). However, we do not believe it significantly impacts the RLC FEC scheme behavior.

Other types of biases may exist that may be visible with smaller tests, for instance to evaluate the convergence speed to a uniform distribution. We therefore perform 200 tests, each of them producing 200 sequences, keeping only the first value of each sequence. We use non-overlapping repair keys for each sequence, starting with value 0 and increasing it after each use.

Value	Min Occurrences	Max Occurrences	Average Occurrences
0	4	21	6.3675
1	4	22	6.0200
2	4	20	6.3125
3	5	23	6.1775
4	5	24	6.1000
5	4	21	6.5925
6	5	30	6.3075
7	6	22	6.2225
8	5	26	6.1750
9	3	21	5.9425
10	5	24	6.3175
11	4	22	6.4300
12	5	21	6.1600

Value	Min Occurrences	Max Occurrences	Average Occurrences
13	5	22	6.3100
14	4	26	6.3950
15	4	21	6.1700

Table 2: tinymt32_rand16() Occurrence Statistics

Table 2 shows across all 200 tests, for each of the 16 possible pseudorandom number values, the minimum (resp. maximum) number of times it appeared in a test, as well as the average number of occurrences across the 200 tests. Although the distribution is not perfect, there is no major bias. On the contrary, in the same conditions, the Park-Miller linear congruential PRNG of [RFC5170] with a result scaled down to 4-bit values, using seeds in sequence starting from 1, systematically returns 0 as the first value during some time. Then, after a certain repair key value threshold, it systematically returns 1, etc.

Evaluation of tinymt32_rand256(): The same approach is used here. Results (not shown) are similar: occurrences vary between 7,810,3368 (i.e., 0.3905%) and 7,814,7952 (i.e., 0.3907%). Here also we see a convergence to the theoretical uniform distribution where each of the 256 possible values would appear exactly 1 / 256 * 100 = 0.390625% of times.

Appendix C. Possible Parameter Derivation (Informational)

Section 3.1 defines several parameters to control the encoder or decoder. This annex proposes techniques to derive these parameters according to the target use-case. This annex is informational, in the sense that using a different derivation technique will not prevent the encoder and decoder to interoperate: a decoder can still recover an erased source symbol without any error. However, in case of a real-time flow, an inappropriate parameter derivation may lead to the decoding of erased source packets after their validity period, making them useless to the target application. This annex proposes an approach to reduce this risk, among other things.

The FEC schemes defined in this document can be used in various manners, depending on the target use-case:

- the source ADU flow they protect may or may not have real-time constraints;
- the source ADU flow may be a Constant Bitrate (CBR) or Variable Bitrate (VBR) flow;
- with a VBR source ADU flow, the flow's minimum and maximum bitrates may or may not be known;
- and the communication path between encoder and decoder may be a CBR communication path (e.g., as with certain LTE-based broadcast channels) or not (general case, e.g., with Internet).

The parameter derivation technique should be suited to the use-case, as described in the following sections.

C.1. Case of a CBR Real-Time Flow

In the following, we consider a real-time flow with max_lat latency budget. The encoding symbol size, E, is constant. The code rate, cr, is also constant, its value depending on the expected communication loss model (this choice is out of scope of this document).

In a first configuration, the source ADU flow bitrate at the input of the FECFRAME sender is fixed and equal to br_in (in bits/s), and this value is known by the FECFRAME sender. It follows that the transmission bitrate at the output of the FECFRAME sender will be higher, depending on the added repair flow overhead. In order to comply with the maximum FEC-related latency budget, we have:

dw_max_size = (max_lat * br_in) / (8 * E)

assuming that the encoding and decoding times are negligible with respect to the target max_lat. This is a reasonable assumption in many situations (e.g., see Section 8.1 in case of small window sizes). Otherwise the max_lat parameter should be adjusted in order to avoid the problem. In any case, interoperability will never be compromised by choosing a too large value.

In a second configuration, the FECFRAME sender generates a fixed bitrate flow, equal to the CBR communication path bitrate equal to br_out (in bits/s), and this value is known by the FECFRAME sender, as in [Roca17]. The maximum source flow bitrate needs to be such that, with the added repair flow overhead, the total transmission bitrate remains inferior or equal to br_out. We have:

dw_max_size = (max_lat * br_out * cr) / (8 * E)

assuming here also that the encoding and decoding times are negligible with respect to the target max_lat.

For decoding to be possible within the latency budget, it is required that the encoding window maximum size be smaller than or at most equal to the decoding window maximum size. The ew_max_size is the main parameter at a FECFRAME sender, but its exact value has no impact on the FEC-related latency budget. The ew_max_size parameter is computed as follows:

ew_max_size = dw_max_size * WSR / 255

In line with [Roca17], WSR = 191 is considered as a reasonable value (the resulting encoding to decoding window size ratio is then close to 0.75), but other values between 1 and 255 inclusive are possible, depending on the use-case.

The dw_max_size is computed by a FECFRAME sender but not explicitly communicated to a FECFRAME receiver. However, a FECFRAME receiver can easily evaluate the ew_max_size by observing the maximum Number of Source Symbols (NSS) value contained in the Repair FEC Payload ID of received FEC Repair Packets (Section 4.1.3). A receiver can then easily compute dw_max_size:

dw_max_size = max_NSS_observed * 255 / WSR

A receiver can then choose an appropriate linear system maximum size:

ls_max_size >= dw_max_size

It is good practice to use a larger value for ls_max_size as explained in Appendix D, which does not impact maximum latency nor interoperability.

In any case, for a given use-case (i.e., for target encoding and decoding devices and desired protection levels in front of communication impairments) and for the computed ew_max_size, dw_max_size and ls_max_size values, it is **RECOMMENDED** to check that the maximum encoding time and maximum memory requirements at a FECFRAME sender, and maximum decoding time and maximum memory requirements at a FECFRAME receiver, stay within reasonable bounds. When assuming that the encoding and decoding times are negligible with respect to the target max_lat, this should be verified as well, otherwise the max_lat **SHOULD** be adjusted accordingly.

The particular case of session start needs to be managed appropriately since the ew_size, starting at zero, increases each time a new source ADU is received by the FECFRAME sender, until it reaches the ew_max_size value. Therefore, a FECFRAME receiver **SHOULD** continuously observe the received FEC Repair Packets, since the NSS value carried in the Repair FEC Payload ID will increase too, and adjust its ls_max_size accordingly if need be. With a CBR flow, session start is expected to be the only moment when the encoding window size will increase. Similarly, with a CBR real-time flow, the session end is expected to be the only moment when the encoding window size will progressively decrease. No adjustment of the ls_max_size is required at the FECFRAME receiver in that case.

C.2. Other Types of Real-Time Flow

In the following, we consider a real-time source ADU flow with a max_lat latency budget and a variable bitrate (VBR) measured at the entry of the FECFRAME sender. A first approach consists in considering the smallest instantaneous bitrate of the source ADU flow, when this parameter is known, and to reuse the derivation of Appendix C.1. Considering the smallest bitrate means that the encoding and decoding window maximum size estimations are pessimistic: these windows have the smallest size required to enable on-time decoding at a FECFRAME receiver. If the instantaneous bitrate is higher than this smallest bitrate, this approach leads to an encoding window that is unnecessarily small, which reduces robustness in front of long erasure bursts.

Another approach consists in using ADU timing information (e.g., using the timestamp field of an RTP packet header, or registering the time upon receiving a new ADU). From the global FEC-related latency budget, the FECFRAME sender can derive a practical maximum latency budget for encoding operations, max_lat_for_encoding. For the FEC schemes specified in this document, this latency budget **SHOULD** be computed with:

max_lat_for_encoding = max_lat * WSR / 255

It follows that any source symbols associated to an ADU that has timed-out with respect to max_lat_for_encoding **SHOULD** be removed from the encoding window. With this approach there is no pre-determined ew_size value: this value fluctuates over the time according to the instantaneous source ADU flow bitrate. For practical reasons, a FECFRAME sender may still require that ew_size does not increase beyond a maximum value (Appendix C.3).

With both approaches, and no matter the choice of the FECFRAME sender, a FECFRAME receiver can still easily evaluate the ew_max_size by observing the maximum Number of Source Symbols (NSS) value contained in the Repair FEC Payload ID of received FEC Repair Packets. A receiver can then compute dw_max_size and derive an appropriate ls_max_size as explained in Appendix C.1.

When the observed NSS fluctuates significantly, a FECFRAME receiver may want to adapt its ls_max_size accordingly. In particular when the NSS is significantly reduced, a FECFRAME receiver may want to reduce the ls_max_size too in order to limit computation complexity. A balance must be found between using an ls_max_size "too large" (which increases computation complexity and memory requirements) and the opposite (which reduces recovery performance).

C.3. Case of a Non-Real-Time Flow

Finally there are configurations where a source ADU flow has no real-time constraints. FECFRAME and the FEC schemes defined in this document can still be used. The choice of appropriate parameter values can be directed by practical considerations. For instance, it can derive from an estimation of the maximum memory amount that could be dedicated to the linear system at a FECFRAME receiver, or the maximum computation complexity at a FECFRAME receiver, both of them depending on the ls_max_size parameter. The same considerations also apply to the FECFRAME sender, where the maximum memory amount and computation complexity depend on the ew_max_size parameter.

Here also, the NSS value contained in FEC Repair Packets is used by a FECFRAME receiver to determine the current coding window size and ew_max_size by observing its maximum value over the time.

Appendix D. Decoding Beyond Maximum Latency Optimization (Informational)

This annex introduces non-normative considerations. It is provided as suggestions, without any impact on interoperability. For more information see [Roca16].

With a real-time source ADU flow, it is possible to improve the decoding performance of Sliding Window Codes without impacting maximum latency, at the cost of extra memory and CPU overhead. The optimization consists, for a FECFRAME receiver, to extend the linear system beyond the decoding window maximum size, by keeping a certain number of old source symbols whereas their associated ADUs timed-out:

ls_max_size > dw_max_size

Usually the following choice is a good trade-off between decoding performance and extra CPU overhead:

ls_max_size = 2 * dw_max_size

When the dw_max_size is very small, it may be preferable to keep a minimum ls_max_size value (e.g., LS_MIN_SIZE_DEFAULT = 40 symbols). Going below this threshold will not save a significant amount of memory nor CPU cycles. Therefore:

ls_max_size = max(2 * dw_max_size, LS_MIN_SIZE_DEFAULT)

Finally, it is worth noting that a receiver that benefits from an FEC protection significantly higher than what is required to recover from packet losses, can choose to reduce the ls_max_size. In that case lost ADUs will be recovered without relying on this optimization.

```
                              ls_max_size
/--------------------------------^------------------------------\

        late source symbols
 (pot. decoded but not delivered)            dw_max_size
/--------------^-----------------\ /--------------^---------------\
src0 src1 src2 src3 src4 src5 src6 src7 src8 src9 src10 src11 src12
```

Figure 11: Relationship between Parameters to Decode beyond Maximum Latency

It means that source symbols, and therefore ADUs, may be decoded even if the added latency exceeds the maximum value permitted by the application (the "late source symbols" of Figure 11). It follows that the corresponding ADUs will not be useful to the application. However, decoding these "late symbols" significantly improves the global robustness in bad reception conditions and is therefore recommended for receivers experiencing bad communication conditions [Roca16]. In any case whether or not to use this optimization and what exact value to use for the ls_max_size parameter are local decisions made by each receiver independently, without any impact on the other receivers nor on the source.

Acknowledgments

The authors would like to thank the three TSVWG chairs, Wesley Eddy (our shepherd), David Black, and Gorry Fairhurst; as well as Spencer Dawkins, our responsible AD; and all those who provided comments -- namely (in alphabetical order), Alan DeKok, Jonathan Detchart, Russ Housley, Emmanuel Lochin, Marie-Jose Montpetit, and Greg Skinner. Last but not least, the authors are really grateful to the IESG members, in particular Benjamin Kaduk, Mirja Kuehlewind, Eric Rescorla, Adam Roach, and Roman Danyliw for their highly valuable feedback that greatly contributed to improving this specification.

Authors' Addresses

Vincent Roca
INRIA
Univ. Grenoble Alpes
France
Email: vincent.roca@inria.fr

Belkacem Teibi
INRIA
Univ. Grenoble Alpes
France
Email: belkacem.teibi@gmail.com