Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: December 21, 2017

JM. Valin
Mozilla Corporation
K. Vos
vocTone

June 19, 2017

# Updates to the Opus Audio Codec
# draft-ietf-codec-opus-update-06

Abstract

This document addresses minor issues that were found in the specification of the Opus audio codec in RFC 6716 [RFC6716].

Table of Contents



## 1. Introduction

This document addresses minor issues that were discovered in the reference implementation of the Opus codec that serves as the specification in RFC 6716 [RFC6716]. Only issues affecting the decoder are listed here. An up-to-date implementation of the Opus encoder can be found at https://opus-codec.org/.

Some of the changes in this document update normative behaviour in a way that requires new test vectors. The English text of the specification is unaffected, only the C implementation is. The updated specification remains fully compatible with the original specification.

Note: due to RFC formatting conventions, lines exceeding the column width in the patch are split using a backslash character. The backslashes at the end of a line and the white space at the beginning of the following line are not part of the patch. A properly formatted patch including all changes is available at <https://jmvalin.ca/misc_stuff/opus_update.patch>. (EDITOR: change to an ietf.org link when ready)

## 2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

## 3. Stereo State Reset in SILK

The reference implementation does not reinitialize the stereo state during a mode switch. The old stereo memory can produce a brief impulse (i.e. single sample) in the decoded audio. This can be fixed by changing silk/dec_API.c at line 72:

```
     for( n = 0; n < DECODER_NUM_CHANNELS; n++ ) {
         ret  = silk_init_decoder( &channel_state[ n ] );
     }
+    silk_memset(&((silk_decoder *)decState)->sStereo, 0,
+                sizeof(((silk_decoder *)decState)->sStereo));
+    /* Not strictly needed, but it's cleaner that way */
+    ((silk_decoder *)decState)->prev_decode_only_middle = 0;

     return ret;
 }
```

This change affects the normative part of the decoder, although the amount of change is too small to make a significant impact on testvectors.

## 4. Parsing of the Opus Packet Padding

It was discovered that some invalid packets of very large size could trigger an out-of-bounds read in the Opus packet parsing code responsible for padding. This is due to an integer overflow if the signaled padding exceeds 2^31-1 bytes (the actual packet may be smaller). The code can be fixed by applying the following changes at line 596 of src/opus_decoder.c:

```
       /* Padding flag is bit 6 */
       if (ch&0x40)
       {
-         int padding=0;
          int p;
          do {
             if (len<=0)
                return OPUS_INVALID_PACKET;
             p = *data++;
             len--;
-            padding += p==255 ? 254: p;
+            len -= p==255 ? 254: p;
          } while (p==255);
-         len -= padding;
       }
```

This packet parsing issue is limited to reading memory up to about 60 kB beyond the compressed buffer. This can only be triggered by a compressed packet more than about 16 MB long, so it's not a problem for RTP. In theory, it _could_ crash a file decoder (e.g. Opus in Ogg) if the memory just after the incoming packet is out-of-range, but our attempts to trigger such a crash in a production application built using an affected version of the Opus decoder failed.

## 5. Resampler buffer

The SILK resampler had the following issues:

1. The calls to memcpy() were using sizeof(opus_int32), but the type of the local buffer was opus_int16.

2. Because the size was wrong, this potentially allowed the source and destination regions of the memcpy() to overlap. We _believe_ that nSamplesIn is at least fs_in_khZ, which is at least 8. Since RESAMPLER_ORDER_FIR_12 is only 8, that should not be a problem once the type size is fixed.

3. The size of the buffer used RESAMPLER_MAX_BATCH_SIZE_IN, but the data stored in it was actually _twice_ the input batch size (nSamplesIn<<1).

The fact that the code never produced any error in testing (including when run under the Valgrind memory debugger), suggests that in practice the batch sizes are reasonable enough that none of the issues above was ever a problem. However, proving that is non-obvious.

The code can be fixed by applying the following changes to line 78 of silk/resampler_private_IIR_FIR.c:

```
 )
 {
     silk_resampler_state_struct *S = \
(silk_resampler_state_struct *)SS;
     opus_int32 nSamplesIn;
     opus_int32 max_index_Q16, index_increment_Q16;
-    opus_int16 buf[ RESAMPLER_MAX_BATCH_SIZE_IN + \
RESAMPLER_ORDER_FIR_12 ];
+    opus_int16 buf[ 2*RESAMPLER_MAX_BATCH_SIZE_IN + \
RESAMPLER_ORDER_FIR_12 ];

     /* Copy buffered samples to start of buffer */
-    silk_memcpy( buf, S->sFIR, RESAMPLER_ORDER_FIR_12 \
* sizeof( opus_int32 ) );
```

```
+    silk_memcpy( buf, S->sFIR, RESAMPLER_ORDER_FIR_12 \
* sizeof( opus_int16 ) );

     /* Iterate over blocks of frameSizeIn input samples */
     index_increment_Q16 = S->invRatio_Q16;
     while( 1 ) {
         nSamplesIn = silk_min( inLen, S->batchSize );

         /* Upsample 2x */
         silk_resampler_private_up2_HQ( S->sIIR, &buf[ \
RESAMPLER_ORDER_FIR_12 ], in, nSamplesIn );

         max_index_Q16 = silk_LSHIFT32( nSamplesIn, 16 + 1 \
);         /* + 1 because 2x upsampling */
         out = silk_resampler_private_IIR_FIR_INTERPOL( out, \
buf, max_index_Q16, index_increment_Q16 );
         in += nSamplesIn;
         inLen -= nSamplesIn;

         if( inLen > 0 ) {
             /* More iterations to do; copy last part of \
filtered signal to beginning of buffer */
-            silk_memcpy( buf, &buf[ nSamplesIn << 1 ], \
RESAMPLER_ORDER_FIR_12 * sizeof( opus_int32 ) );
+            silk_memmove( buf, &buf[ nSamplesIn << 1 ], \
RESAMPLER_ORDER_FIR_12 * sizeof( opus_int16 ) );
         } else {
             break;
         }
     }

     /* Copy last part of filtered signal to the state for \
the next call */
-    silk_memcpy( S->sFIR, &buf[ nSamplesIn << 1 ], \
RESAMPLER_ORDER_FIR_12 * sizeof( opus_int32 ) );
+    silk_memcpy( S->sFIR, &buf[ nSamplesIn << 1 ], \
RESAMPLER_ORDER_FIR_12 * sizeof( opus_int16 ) );
 }
```

## 6. Integer wrap-around in inverse gain computation

It was discovered through decoder fuzzing that some bitstreams could produce integer values exceeding 32-bits in LPC_inverse_pred_gain_QA(), causing a wrap-around. Although the error is harmless in practice, the C standard considers the behavior as undefined, so the following patch to line 87 of silk/LPC_inv_pred_gain.c detects values that do not fit in a 32-bit integer and considers the corresponding filters unstable:

```
          /* Update AR coefficient */
          for( n = 0; n < k; n++ ) {
  -           tmp_QA = Aold_QA[ n ] - MUL32_FRAC_Q( \
  Aold_QA[ k - n - 1 ], rc_Q31, 31 );
  -           Anew_QA[ n ] = MUL32_FRAC_Q( tmp_QA, rc_mult2 , mult2Q );
  +           opus_int64 tmp64;
  +           tmp_QA = silk_SUB_SAT32( Aold_QA[ n ], MUL32_FRAC_Q( \
  Aold_QA[ k - n - 1 ], rc_Q31, 31 ) );
  +           tmp64 = silk_RSHIFT_ROUND64( silk_SMULL( tmp_QA, \
  rc_mult2 ), mult2Q);
  +           if( tmp64 > silk_int32_MAX || tmp64 < silk_int32_MIN ) {
  +              return 0;
  +           }
  +           Anew_QA[ n ] = ( opus_int32 )tmp64;
          }
```

## 7. Integer wrap-around in LSF decoding

It was discovered -- also from decoder fuzzing -- that an integer wrap-around could occur when decoding line spectral frequency coefficients from extreme bitstreams. The end result of the wrap-around is an illegal read access on the stack, which the authors do not believe is exploitable but should nonetheless be fixed. The following patch to line 137 of silk/NLSF_stabilize.c prevents the problem:

```
            /* Keep delta_min distance between the NLSFs */
          for( i = 1; i < L; i++ )
  -           NLSF_Q15[i] = silk_max_int( NLSF_Q15[i], \
  NLSF_Q15[i-1] + NDeltaMin_Q15[i] );
  +           NLSF_Q15[i] = silk_max_int( NLSF_Q15[i], \
  silk_ADD_SAT16( NLSF_Q15[i-1], NDeltaMin_Q15[i] ) );

          /* Last NLSF should be no higher than 1 - NDeltaMin[L] */
```

## 8. Cap on Band Energy

On extreme bit-streams, it is possible for log-domain band energy levels to exceed the maximum single-precision floating point value once converted to a linear scale. This would later cause the decoded values to be NaN, possibly causing problems in the software using the PCM values. This can be avoided with the following patch to line 552 of celt/quant_bands.c:

```
         {
            opus_val16 lg = ADD16(oldEBands[i+c*m->nbEBands],
                                  SHL16((opus_val16)eMeans[i],6));
+           lg = MIN32(QCONST32(32.f, 16), lg);
            eBands[i+c*m->nbEBands] = PSHR32(celt_exp2(lg),4);
         }
         for (;i<m->nbEBands;i++)
```

## 9. Hybrid Folding

When encoding in hybrid mode at low bitrate, we sometimes only have enough bits to code a single CELT band (8 - 9.6 kHz). When that happens, the second band (CELT band 18, from 9.6 to 12 kHz) cannot use folding because it is wider than the amount already coded, and falls back to LCG noise. Because it can also happen on transients (e.g. stops), it can cause audible pre-echo.

To address the issue, we change the folding behavior so that it is never forced to fall back to LCG due to the first band not containing enough coefficients to fold onto the second band. This is achieved by simply repeating part of the first band in the folding of the second band. This changes the code in celt/bands.c around line 1237:

```
           b = 0;
        }

-       if (resynth && M*eBands[i]-N >= M*eBands[start] && \
(update_lowband || lowband_offset==0))
+       if (resynth && (M*eBands[i]-N >= M*eBands[start] || \
i==start+1) && (update_lowband || lowband_offset==0))
            lowband_offset = i;

+       if (i == start+1)
+       {
+          int n1, n2;
+          int offset;
+          n1 = M*(eBands[start+1]-eBands[start]);
+          n2 = M*(eBands[start+2]-eBands[start+1]);
+          offset = M*eBands[start];
+          /* Duplicate enough of the first band folding data to \
be able to fold the second band.
+             Copies no data for CELT-only mode. */
+          OPUS_COPY(&norm[offset+n1], &norm[offset+2*n1 - n2], n2-n1);
+          if (C==2)
+             OPUS_COPY(&norm2[offset+n1], &norm2[offset+2*n1 - n2], \
n2-n1);
+       }
+
        tf_change = tf_res[i];
        if (i>=m->effEBands)
        {
```

as well as line 1260:

```
           fold_start = lowband_offset;
           while(M*eBands[--fold_start] > effective_lowband);
           fold_end = lowband_offset-1;
-          while(M*eBands[++fold_end] < effective_lowband+N);
+          while(++fold_end < i && M*eBands[fold_end] < \
effective_lowband+N);
           x_cm = y_cm = 0;
           fold_i = fold_start; do {
             x_cm |= collapse_masks[fold_i*C+0];
```

The fix does not impact compatibility, because the improvement does not depend on the encoder doing anything special. There is also no reasonable way for an encoder to use the original behavior to improve quality over the proposed change.

## 10. Downmix to Mono

The last issue is not strictly a bug, but it is an issue that has been reported when downmixing an Opus decoded stream to mono, whether this is done inside the decoder or as a post-processing step on the stereo decoder output. Opus intensity stereo allows optionally coding the two channels 180-degrees out of phase on a per-band basis. This provides better stereo quality than forcing the two channels to be in phase, but when the output is downmixed to mono, the energy in the affected bands is cancelled sometimes resulting in audible artefacts.

As a work-around for this issue, the decoder MAY choose not to apply the 180-degree phase shift when the output is meant to be downmixed (inside or outside of the decoder).

## 11. New Test Vectors

Changes in Section 9 and Section 10 have sufficient impact on the testvectors to make them fail. For this reason, this document also updates the Opus test vectors. The new test vectors now include two decoded outputs for the same bitstream. The outputs with suffix 'm' do not apply the CELT 180-degree phase shift as allowed in Section 10, while the outputs without the suffix do. An implementation is compliant as long as it passes either set of vectors.

In addition, any Opus implementation that passes the original test vectors from RFC 6716 [RFC6716] is still compliant with the Opus specification. However, newer implementations SHOULD be based on the new test vectors rather than the old ones.

The new test vectors are located at <https://jmvalin.ca/misc_stuff/opus_newvectors.tar.gz>. (EDITOR: change to an ietf.org link when ready)

## 12. IANA Considerations

This document makes no request of IANA.

Note to RFC Editor: this section may be removed on publication as an RFC.

## 13. Acknowledgements

We would like to thank Juri Aedla for reporting the issue with the parsing of the Opus padding. Also, thanks to Jonathan Lennox and Mark Harris for their feedback on this document.

## 14. References

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

[RFC6716] Valin, JM., Vos, K., and T. Terriberry, "Definition of the Opus Audio Codec", RFC 6716, DOI 10.17487/RFC6716, September 2012, <http://www.rfc-editor.org/info/rfc6716>.

Authors' Addresses

Jean-Marc Valin
Mozilla Corporation
331 E. Evelyn Avenue
Mountain View, CA 94041
USA

Phone: +1 650 903-0800
Email: jmvalin@jmvalin.ca

Koen Vos
vocTone

Email: koenvos74@gmail.com